AFRL-RI-RS-TR-2015-209

PROSPECTS FOR EVIDENCE-BASED SOFTWARE ASSURANCE: MODELS AND ANALYSIS

CARNEGIE MELLON UNIVERSITY

SEPTEMBER 2015

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2015-209 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
WILLIAM E. MCKEEVER JR.
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government’s approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* SEP 2015	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* MAR 2012 – MAR 2015

4. TITLE AND SUBTITLE PROSPECTS FOR EVIDENCE-BASED SOFTWARE ASSURANCE: MODELS AND ANALYSIS	**5a. CONTRACT NUMBER** FA8750-12-2-0139
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62788F
6. AUTHOR(S) William Scherlis	**5d. PROJECT NUMBER** T2UN
	5e. TASK NUMBER CM
	5f. WORK UNIT NUMBER US
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie Mellon University 5000 Forbes Ave Pittsburgh PA 15213	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR’S REPORT NUMBER AFRL-RI-RS-TR-2015-209

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The project addresses the challenge of software assurance in the presence of rich supply chains. As a consequence of the focus on supply chains, the project addresses two broad technical questions: First, what are the elements of an evidence-based approach, relying on both formal and informal evidence that can support assurance judgments that are effective and rapid? Second, How can these ideas support composition of judgments about the many separate software components, libraries, and frameworks that are typically required for larger software projects? Progress on these two questions is intended to inform a broader question of great significance to DoD, which is what are forms of a “software deliverable” that are more effective in support both acceptance evaluation (OT&E) and also ongoing evolution as part of the process of sustainment and modernization. The idea is that a useful body of evidence can link deliverable code and documentation with requirements, architecture, and quality models. Additionally, these models and traceability links can support agile-style evolution in a code

15. SUBJECT TERMS
software assurance, evidence-based software, software supply chain, assurance case, sandboxing, adaptive software

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** **WILLIAM E. MCKEEVER, JR.**
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	SAR	147	**19b. TELEPHONE NUMBER (*Include area code*)** **N/A**

Prospects for evidence-based software assurance: models and analysis

Final Technical Report – September 2015

William L. Scherlis, PI (scherlis@cmu.edu)
Jonathan Aldrich, co-PI (jonathan.aldrich@cs.cmu.edu)

Institute for Software Research (ISR)
Carnegie Mellon University School of Computer Science (SCS)

Table of Contents

1.0 SUMMARY

This report summarizes the goals, research approach, technical results, and community impact of the project, *Prospects for evidence-based software assurance: models and analysis*. The project focused on assessing prospects and advancing the potential for more aggressive evidence-based approaches to supporting the development of confident assurance judgments for complex software. The approaches generally rely on a combination of engineering levers: (1) architecture and design, (2) component management, (3) semantics-based models and technical analyses, (4) prototype tooling. In addition to the technical results described in this report, the team supported workshops and technical discussions among industry and government technical leaders focused on assurance.

2.0 INTRODUCTION

The project addresses the challenge of software assurance in the presence of rich supply chains. As a consequence of the focus on supply chains, the project addresses two broad technical questions: First, what are the elements of an *evidence-based approach*, relying on both formal and informal evidence, that can support assurance judgments that are effective and rapid? Second, how can these ideas support *composition of judgments* about the many separate software components, libraries, and frameworks that are typically required for larger software projects?

Progress on these two questions is intended to inform a broader question of great significance to DoD, which is what are forms of a “software deliverable” that are more effective in support both acceptance evaluation (such as DoD’s Operational Test and Evaluation (OT&E)) and also ongoing evolution as part of the process of sustainment and modernization. The idea is that a useful body of evidence can link deliverable code and documentation with requirements, architecture, and quality models. Additionally, these models and traceability links can support agile-style evolution in a code base.

This assessment effort undertook to consider several technical topics, including (1) models of intended behavior, including diverse quality attributes, structural attributes, and security-related attributes, (2) tools and environments for managing code and models of aspects of design intent, (3) tools to manage formal and informal evaluation elements, including results of static and dynamic analysis, test cases, and inspections, (4) capability to construct and manage traceability linkages in an environment of constant change and evolution of software code and models of intent associated with that code.

Case studies related to larger-scale embedded software were explored, including secure web services.

We report here on progress related to these topics.

2.1. Project Team

The project team included faculty and PhD students:

- William L. Scherlis, PI – Professor of Computer Science and Director of the Institute for Software Research in the Carnegie Mellon School of Computer Science
- Jonathan Aldrich, co-PI – Associate Professor of Computer Science and Director of the PhD program in Software Engineering in the Carnegie Mellon School of Computer Science
- David Garlan – Professor of Computer Science and Director of Software Professional Programs in the Institute for Software Research in the Carnegie Mellon School of Computer Science
- Josh Sunshine – Systems Scientist in the Institute for Software Research in the Carnegie Mellon School of Computer Science
- Michael Maass – PhD student in the PhD program in Software Engineering in the Carnegie Mellon School of Computer Science
- Darya Kurilova – PhD student in the PhD program in Software Engineering in the Carnegie Mellon School of Computer Science

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

The project team examined five specific technical aspects of the two key questions (evidence production and composition of judgments regarding components). In each case, the team considered the kinds of models and analyses that might lead to high levels of assurance – and effective assurance evaluation practices. This included the kinds of evidence that might be manifest in the models and the analyses, as well as the kinds of traceability that might be created to support evaluation practice.

The five technical aspects are part of an overall strategy that recognizes the necessity of working at the architecture level in order to develop scalable approaches to assurance and evidence production. Scalability is almost always a consequence of composability. Composability derives from effective consideration of the design of interfaces and components within architectural models.

Composability means that conclusions regarding components can be combined to reach conclusions regarding assemblies of components. Conclusions regarding components may come from direct models and analysis. But often components are opaque or of such complexity that effective conclusions cannot be reached directly. In these case, approaches based on isolation or sandboxing may be most effective from an engineering cost perspective. This is the first area of consideration in this research.

The quality assertions made regarding components and systems encompass many kinds of quality attributes. Mission context is a primary determiner of the weighting of these attributes in an overall engineering process. But there are also engineering tradeoffs among quality attributes that influence outcomes. These tradeoffs are the second area of consideration in this research.

Interfaces within systems have many dimensions of design beyond the type abstractions, exposed methods, and callbacks. These can include, particularly, complex protocols of interaction across these interfaces, for example including a sequence of actions such as object creation, initialization and configuration, registration for callback events, correct response to events, deregistration for events, and teardown and release of resources. Small defects in the implementation by a client of protocol requirements can lead to security and reliability errors. This is the third area of consideration in this research.

For nearly all systems, much of the code that is part of the system is actually library code written by vendors and other third parties. A fourth area of consideration is adding specifications to library interface definitions that can express more than just type information.

Finally, we note that modern architectures are increasingly dynamic and adaptive. Instead of a static identification of components and structure of interconnection, components may come and go, and topologies of interconnection change in response to external stimuli. The fifth area of consideration is an addressing of this dynamism.

The five areas of consideration all relate to challenges of components and structure and assurance. Below are further details concerning the work undertaken in these areas:

- **Isolation of untrusted components**. To support assurance judgments regarding a software component, there are three basic options: (1) static analysis that leads to verification conclusions, (2) dynamic analysis, monitoring, and logging, that can provide runtime checks of correct (safe and secure) behavior, and (3) encapsulation and isolation of the component ("sandboxing") in order to insulate the remaining parts of the system from any potential undesired behaviors of the component. Sandboxing is a common technique in certain specific applications, such as web browsers. But, surprisingly, there is no systematic analysis of the approach for broader use.

  Our procedure in addressing this was to consider the safe reading of potentially malicious PDF files. Our method is to develop design principles, construct prototypes, and evaluate experimental deployment of those prototypes.

- **Architecture quality tradeoffs**. Assurance judgments are derived from consideration of multiple quality attributes. The weighting among these attributes derives from mission context – in what context and for what purpose is the system to be used, and what are the critical quality attributes to that context. This means that trade-off analysis is a key feature of the engineering process in support of mission-assured systems.

  We approach this issue by considering the nature of the models that might assist in addressing trade-off issues. The models are supported by tools both that allow exploration of the engineering trades and that support accretion of evidence in support of the conclusions reached.

- **Design intent capture**. While modern systems are more component-based and with richer supply chains, the interfaces (Application Programming Interfaces (APIs)) are highly complex, with detailed protocol requirements and pre-conditions on inputs. The OpenSSL (Heartbleed) vulnerability is an example of a failure to address this complexity in a correct manner.

  We approach this complexity challenge through the development of modeling formalisms and tooling to model the protocol issues. This is informed through empirical understanding of how developers approach these complexity issues – and how explicit tool-supported models could assist.

- **Safe use of libraries**. As noted immediately above, modern component interfaces have greater complexity. Indeed, it can be remarked that with modern systems we have achieved a success in reuse (with the ubiquitous use of libraries, frameworks, generators, and common components) but a failure of modularity (with complex protocols, input requirements, and other usage constraints). A challenge is in how these components interact – and, in particular, assuring that the interactions are safe, i.e., compositional.

We approach this issue through exploration of the interplay of type and state attributes of objects. For example, we may want to restrict an input parameter to a **read** method to be a **FileHandle** object that is in the **open** state. Another example considered is in the safe use of HTML5.

- **Assurance for self-adaptive systems**. Self-adaptiveness is becoming an essential feature of modern systems on the basis of requirements for resiliency, autonomy, and the security concept of *moving target defense*. Self-adaptiveness creates challenges for assurance, since executable code – and its architectural structures – may not be statically determinable. That is, architectural models become fluid and changeable. The assurance judgment must then apply to the full family of potential executables and architectural structures, even when this set is very large (as it should be for effective resilience and moving target response).

  The approach taken to this part of the effort is to address some particular technical challenges related to composition, but also to engage the broader community through workshop interactions.

In addition to these five areas of technical consideration, the PI was involved in a number of outreach efforts to agency decision-makers address the full scope of the assurance challenge facing national security systems. This included engagement with multiple NITRD working groups, as noted below.

4.0 RESULTS AND DISCUSSION

The following specific research issues were considered. In each, significant results are identified and associated publications cited.

4.1. Isolation and sandboxing.

When components are opaque, or otherwise resist modeling and analysis in support of composition, they can nonetheless be included in systems safely when they are placed in “containers” that can isolate them from trusted portions of the overall system. This idea of sandboxing is commonplace, for example, in web browser implementations to support both JavaScript and Java. It is also the concept behind the hardware-supported process separation mechanisms intrinsic to all modern operating systems. Many of the recently publicized vulnerabilities pertaining to Java relate to sandbox models and implementations.

We completed an experimental deployment of our in-nimbo sandbox for Adobe Reader at a large aerospace company [1]. Our collaborator was pleased with the performance of the sandbox both in terms of speed and protection it provides. User perception of the performance between in-situ Adobe Reader and a cloud-deployed Reader turns out not to be significant. Additionally, the "chroming" of the in-nimbo Reader is now very similar to the in-situ Reader, with the result that there is a high level of user acceptance. They are now working on transferring the sandbox into practice by polishing a few minor engineering aspects that will make the sandbox easier for typical users to use on the day to day.

Sandboxes impose a security policy, isolating applications and their components from the rest of a system. While many sandboxing techniques exist, state of the art sandboxes generally perform their functions within the system that is being defended. As a result, when the sandbox fails or is bypassed, the security of the surrounding system can no longer be assured. We experiment with the idea of innimbo sandboxing, encapsulating untrusted computations away from the system we are trying to protect [1]. The idea is to delegate computations that may be vulnerable or malicious to virtual machine instances in a cloud computing environment.

This may not reduce the possibility of an in-situ sandbox compromise, but it could significantly reduce the consequences should that possibility be realized. To achieve this advantage, there are additional requirements, including: (1) A regulated channel between the local and cloud environments that supports interaction with the encapsulated application, (2) Performance design that acceptably minimizes latencies in excess of the in-situ baseline.

To test the feasibility of the idea, we built an in-nimbo sandbox for Adobe Reader, an application that historically has been subject to significant attacks. We undertook a prototype deployment with PDF (Portable Document Format) users in a large aerospace firm. In addition to thwarting several examples of existing PDF-based malware, we found that the added increment of latency, perhaps surprisingly, does not overly impair the user experience with respect to performance or usability. We consider this to be a successful tech transfer. We are considering possible mechanisms to transition this result into more widespread use.

4.2. Architecture design tradeoffs

We investigated automated support for making tradeoffs in system configurations that balance the quality attributes of timeliness and fidelity (which are typically in conflict) [2]. Specifically we developed tools that automatically configure workflow architectures to meet timeliness guarantees, based on user inputs that select possible dimensions of fidelity reduction. The tools explore the space of possible configurations, and select the one that meets the timing requirements, while best satisfying the fidelity preferences.

In many scientific fields, simulations and analyses require compositions of computational entities such as web services, programs, and applications. In such fields, users may want various trade-offs between different qualities. Examples include: (i) performing a quick approximation vs. an accurate, but slower, experiment, (ii) using local slower execution environments vs. remote, but advanced, computing facilities, (iii) using quicker approximation algorithms vs. computationally expensive algorithms with smaller data.

However, such trade-offs are difficult to make as many such decisions today are either (a) wired into a fixed configuration and cannot be changed, or (b) require detailed systems knowledge and experimentation to determine what configuration to use. We propose an approach that uses architectural models coupled with automated design space generation for making fidelity and timeliness trade-offs. The work illustrated this approach through an example in the intelligence analysis domain [2].

4.3. Design intent capture

We explored whether evidence-based assurance tools that are structured like types, in that they document small, composable pieces of design intent that can be statically checked, can provide productivity benefits in addition to assurance. A positive result in this area would make it easier to adopt assurance tools, as developers could be confident that using the tools would be likely to speed up development rather than slow it down. In an ECOOP 2014 (European Conference on Object Oriented Programming) paper we describe a lab study showing that documentation that can be generated from type-like state specifications makes developers more productive when performing state-related software development tasks [3].

Application Programming Interfaces (APIs) often define object protocols. Objects with protocols have a finite number of states and in each state a different set of method calls is valid. Many researchers have developed protocol verification tools because protocols are notoriously difficult to follow correctly. However, recent research suggests that a major challenge for API protocol programmers is effectively searching the state space. Verification is an ineffective guide for this kind of search.

The research instead proposes Plaiddoc [3], which is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. The research compares Plaiddoc to a Javadoc control in a between-subjects laboratory experiment. It found that Plaiddoc participants complete state search tasks in significantly less time and with significantly fewer errors than Javadoc participants.

An important aspect of this work is the connection to real-world software development tasks. The tasks used in the study above were gleaned from questions on StackOverflow and were studied qualitatively in the laboratory. In this study, we discovered a number of barriers programmers face when working with stateful libraries, informing the design of future assurance tools [4].

As noted above, APIs often define protocols — restrictions on the order of client calls to API methods. API protocols are common and difficult to use, which has generated tremendous research effort in alternative specification, implementation, and verification techniques. However, little is understood about the barriers programmers face when using these APIs, and therefore the research effort may be misdirected.

To understand these barriers better, the researchers performed a two-part qualitative study [4]. The first part was a study of developer forums to identify problems that developers have with protocols. The second part was a think-aloud observational study, in which the struggle of professional programmers with these same problems was observed to get more detail on the nature of their struggles and how they use available resources. The researchers concluded that programmer time was spent primarily on four types of searches of the protocol state space. These observations suggest protocol-targeted tools, languages, and verification techniques will be most effective if they enable programmers to efficiently perform state search.

4.4. Safe use of libraries

We made substantial progress on a project to support libraries that can extend the syntax and semantics of a language in composable ways. The goal is users of libraries and frameworks can use abstractions defined in those libraries and frameworks to express their programs and associated design intent in more direct ways, and the libraries and frameworks can define custom analyses that provide evidence-based assurance that the abstractions are used safely and correctly. We developed an initial type theory that describes our approach and formally verified that our mechanism is compositional, so that separately-developed libraries and frameworks can easily be used together [5].

Programming languages often include specialized syntax for common datatypes (e.g. lists) and some also build in support for specific specialized datatypes (e.g. regular expressions), but user-defined types must use general-purpose syntax. Frustration with this causes developers to use strings, rather than structured data, with alarming frequency, leading to correctness, performance, security, and usability issues. Allowing library providers to modularly extend a language with new syntax could help address these issues. Unfortunately, prior mechanisms either limit expressiveness or are not safely composable: individually unambiguous extensions can still cause ambiguities when used together. The research introduces type-specific languages (TSLs): logic associated with a type that determines how the bodies of generic literals, able to contain arbitrary syntax, are parsed and elaborated, hygienically. The TSL for a type is invoked only when a literal appears where a term of that type is expected, guaranteeing noninterference. Evidence is given to support the applicability of this approach and formally specify it with a bidirectionally typed elaboration semantics for the Wyvern programming language [6].

Modern software relies on the use of libraries, but assuring the correct use of those libraries, including the order of operations, can be challenging. We are investigating a new kind of compositional abstraction that includes both the behavior of an object (e.g., the sequence of method calls to it) and the way the object is shared (e.g., if we have the only pointer to it, or if the object is used as to communicate between different threads or parts of the program). Unlike prior work by us and others, the sharing and behavior is part of a single abstraction that can be manipulated algebraically as the object is aliased or as operations are performed on it [6]. This improves compositionality and expressiveness relative to prior typestate assurance approaches.

The use of shared mutable state, commonly seen in object-oriented systems, is often problematic due to the potential conflicting interactions between aliases to the same state. The team presented a substructural type system outfitted with a novel lightweight interference control mechanism, rely-guarantee protocols, that enables controlled aliasing of shared resources [6]. By assigning each alias separate roles, encoded in a novel protocol abstraction in the spirit of rely-guarantee reasoning, our type system ensures that challenging uses of shared state will never interfere in an unsafe fashion. In particular, rely-guarantee protocols ensure that each alias will never observe an unexpected value, or type, when inspecting shared memory regardless of how the changes to that shared state (originating from potentially unknown program contexts) are interleaved at run-time.

In related work, safe HTML5 subsets were considered as a way to provide a means to avoid certain exploits [1]. Websites often allow users to enter a subset of HTML as input that will be reflected back to the browser in a state that can still be rendered. This is done for many reasons, such as allowing third parties to: format posts on hosted blogs, layout equations and code on question-and-answer forums, create rich auction listings, display ads, etc. Websites that use unsafe subsets of HTML in these cases may allow third parties to introduce XSS (cross-site scripting), CSRF (cross-site request forgery), or Clickjacking exploits. Picking a useful, safe subset of HTML is hard because the semantics of web languages are often vague and they interact in ways that can be subtle and dangerous. We methodically surveyed the HTML5 specification and identified precisely the cases where XSS, CSRF, and clickjacking exploits can appear [1]. This is an important step forward in defining a provably safe subset of HTML.

4.5. Assurance for self-adaptive systems

We helped organize, and participated in, a Dagstuhl **Workshop on Assurances of Self-Adaptive Systems** held December 15-19, 2013 in Germany [. This workshop brought together 43 prominent researchers in the field of self-adaptive systems to (a) characterize the state of the art in assurance for such systems [7], and (b) provide roadmaps for research in advancing our ability to gain confidence in systems that adapt themselves at run time[8].

4.6. Community interactions

The PI co-organized a workshop focused on Designed-In Security as developed by the Networking, Information Technology Research and Development (NITRD) Cybersecurity and Information Assurance (CSIA) working group (https://www.nitrd.gov/cybersecurity/#DIS). Preliminary results of the workshop were jointly presented by the PI at the Cybersecurity Innovation Forum (sponsored by NSA and NIST) in Baltimore in January 2014. DIS is identified by NITRD as one of the five critical areas for “cybersecurity game-change R&D.” Supply-chain and composition technical issues loom large, as do practicability and ROI issues related to modeling, tool use, and deep analysis. The PI co-authored the report (sponsored jointly by NITRD and SCORE) and led a panel discussion at the NITRD HCSS Conference in May 2014. The report (http://cps-vo.org/node/12673) and panel description are available at the cps-vo (http://cps-vo.org/taxonomy/term/6694). There were three parallel workshop sessions, focused on Business Case, Software, and Hardware; the PI led the Software workshop session and was the lead author for the sections related to software.

> *Executive Summary*. In addressing the need for improved Designed-In Security (DIS) research and practice, the Federal Networking and Information Technology Research and Technology (NITRD) Program and the Special Cyber Operations Research and Engineering Interagency Working Group (SCORE IWG) have begun conducting a series of small, invitational workshops with the aim of placing leading researchers in direct contact with leading practitioners to ensure that future research targets those underlying problems that truly limit the practice. NITRD and SCORE have adopted a multidisciplinary approach whereby a broad range of experts are included in the workshops to address the various problems and solutions that may have previously gone overlooked as a consequence of an overly narrow focus. The workshops offer an opportunity for practitioners to become more familiar with research concepts to address their current needs and, similarly, for academics to gain familiarity with operational challenges as well as better identify educational needs and approaches in building a workforce capable of designing and producing higher assurance systems. The innovative ideas identified in the first workshop will be developed and evaluated in subsequent workshops of the series.

The PI also made multiple presentations related to software assurance to various NITRD Working Groups including SDP (Software Design and Productivity), HCSS (High Confidence Software and Systems), and CSIA (Cybersecurity and Information Assurance).

4.7. Technology transition

The sandboxing technology described above is the subject of a Provisional Application for a U.S. Patent from Carnegie Mellon (William Scherlis, Jonathan Aldrich, Michael Maass), filed in 2013 (CMU Docket #2013-083).

5.0 CONCLUSIONS

The project, “*Prospects for evidence-based software assurance: models and analysis*,” addresses the challenge of software assurance in the presence of rich supply chains. Modern systems more consist of multiple components drawn from diverse sources, including libraries, frameworks, generators, and specific functional subsystems. With this beneficial modularity comes a complexity of interfaces and often steep gradients of trust within a system, with the possibility of attack surfaces within a system. There are significant engineering benefits, but also increased difficulty in reaching confident assurance judgments.

As noted, the project addressed two broad technical questions: First, what are the elements of an evidence-based approach, relying on both formal and informal evidence, that can support assurance judgments that are effective and rapid? Second, how can these ideas support composition of judgments about the many separate software components, libraries, and frameworks that are typically required for larger software projects?

The project presented results in several top publication venues:

- Six published papers in diverse venues
- A Distinguished Paper award
- A provisional patent application
- A field trial at Boeing for a software prototype
- A Dagstuhl workshop on self-adaptive systems
- A joint SCORE and NITRD workshop on Designed-In Security
- Several presentations to NITRD Working Groups including SDP (Software Design and Productivity), HCSS (High Confidence Software and Systems), and CSIA (Cybersecurity and Information Assurance)
- Two additional papers still in submission.

6.0 REFERENCES

[1] Michael Maass, Jonathan Aldrich, and William Scherlis. In-Nimbo Sandboxing. *Proceedings of the Science of Security Conference (HotSOS)*, 2014.

[2] Vishal Dwivedi, David Garlan, Jürgen Pfeffer and Bradley Schmerl. Model-based Assistance for Making Time/Fidelity Trade-offs in Component Compositions. In *11th International Conference on Information Technology: New Generations (ITNG 2014),* Special track on: Model-Driven, Component-Based Software Engineering (MDCBSE), Las Vegas, NV, 7-9 April 2014.

[3] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming. In *European Conference on Object-Oriented Programming*, 2014.

[4] Joshua Sunshine and Jonathan Aldrich. Searching the State Space: A Qualitative Study of API Protocol Usability. In *International Conference on Program Comprehension (ICPC)*, 2015.

[5] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely Composable Type-Specific Languages. Submitted to the European Conference on Object-Oriented Programming, 2014.

[6] Rely-Guarantee Protocols. Filipe Militao, Jonathan Aldrich, and Luis Caires. Submitted to the European Conference on Object-Oriented Programming, 2014.

[7] Bradley Schmerl, Jesper Andersson, Thomas Vogel, Myra B. Cohen, Cecilia M. F. Rubira, Yuriy Brun, Alessandra Gorla, Franco Zambonelli, and Luciano Baresi. Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems. (2013). Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511). Dagstuhl Reports

[8] D. Weyns, N. Bencomo, R. Calinescu, J. Camara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J.M. Jezequel, S. Malek, R. Mirandola, M. Mori, and G. Tamburrelli. Perpetual Assurances in Self-Adaptive Systems. (2013). Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511). Dagstuhl Reports

[9] Celia Merzbacher, Ronald Perez, William Scherlis, Tomas Vagoun, Claire Vishik, Angelos Keromytis, Lisa Coote, et. al. Workshop Report, Designing In Security: Current Practices and Research Needs, NITRD and SCORE Workshop Series July 2013

7.0 ACRONYMS

API	Application Program Interface
CSIA	Cyber Security and Information Assurance
CSRF	Cross-Site Request Forgery
DIS	Design In Security
ECOOP	European Conference on Object-Oriented Programming
HCSS	High Confidence Software and Systems
HTML	Hyper Text Markup Language
IWG	Interagency Working Group
NITRD	Networking and Information Technology Research and Development
PDF	Portable Document Format
PI	Principal Investigator
SCORE	Special Cyber Operations Research and Engineering
SDP	Software Design and Productivity
SSL	Secure Sockets Layer
TSL	Type Specific Language
XSS	Cross Site Scripting

APPENDIX A: PUBLICATIONS

In-Nimbo Sandboxing

Michael Maass, William L. Scherlis, and Jonathan Aldrich
Institute for Software Research
Carnegie Mellon University
{mmaass, wls, aldrich}@cs.cmu.edu

ABSTRACT

Sandboxes impose a security policy, isolating applications and their components from the rest of a system. While many sandboxing techniques exist, state of the art sandboxes generally perform their functions within the system that is being defended. As a result, when the sandbox fails or is bypassed, the security of the surrounding system can no longer be assured. We experiment with the idea of in-nimbo sandboxing, encapsulating untrusted computations away from the system we are trying to protect. The idea is to delegate computations that may be vulnerable or malicious to virtual machine instances in a cloud computing environment.

This may not reduce the possibility of an in-situ sandbox compromise, but it could significantly reduce the consequences should that possibility be realized. To achieve this advantage, there are additional requirements, including: (1) A regulated channel between the local and cloud environments that supports interaction with the encapsulated application, (2) Performance design that acceptably minimizes latencies in excess of the in-situ baseline.

To test the feasibility of the idea, we built an in-nimbo sandbox for Adobe Reader, an application that historically has been subject to significant attacks. We undertook a prototype deployment with PDF users in a large aerospace firm. In addition to thwarting several examples of existing PDF-based malware, we found that the added increment of latency, perhaps surprisingly, does not overly impair the user experience with respect to performance or usability.

1. INTRODUCTION

Sandboxes are the most common way to secure systems and components that are currently intractable to verify and that we cannot trust. Application sandboxing is a technique used to impose a security policy on an application. Rather than assuring the compliance of an application or a computation with a policy, a sandbox is constructed to encapsulate the application or computation and any malicious behavior it may manifest. The sandbox implements a security policy on control and data flows to reduce the likelihood and consequences to a target system of malicious code execution within the sandbox. Sandboxing is also useful for increasing confidence in the execution of applications that are trusted but that are nonetheless vulnerable, due to complexity or other factors.

But what happens when the sandbox fails? This paper presents and evaluates an application-focused sandboxing technique that is intended to address both sides of the risk calculation – mitigating the consequences of traditional sandbox failures while also increasing the effort required by an attacker attempting to compromise the target system. Our technique is reminiscent of software as a service, thus allowing us to evaluate the security benefits of those and similar architectures. We present the technique, describe a prototype we developed to support a field trial deployment, and assess the technique according to a set of defined criteria. Here is a summary of our hypotheses regarding the in-nimbo technique for application sandboxing:

- Attack Surface Design Flexibility: In-nimbo sandboxing provides flexibility in attack surface design. We focus on tailoring the sandbox to the application, which doesn't allow for a "one size fits all" implementation. Our technique allows architects to more easily design and implement an attack surface they can confidently defend when compared to other techniques. This is because the technique is less constrained by structures within an existing client system.
- Attack Surface Extent: Our technique results in encapsulated components with smaller, more defensible attack surfaces compared to the cases where the component is encapsulated using other techniques. Along with the previous criterion, this should have the effect of diminishing the "likelihood" part of the risk product.
- Consequence of Attack Success: Remote encapsulation reduces the consequences of attack success. Our technique reduces the magnitude of the damage resulting from an attack on the encapsulated component when compared to the same attack on the component when it is encapsulated using other techniques. That is, we suggest that our approach diminishes the extent of consequence in the risk product.

We also apply the following criteria:

- Performance: We focus on latency and ignore resource consumption. Our technique slightly increases the user-

HotSoS '14 Raleigh, NC USA
Copyright 2014 ACM 978-1-4503-2907-1/14/04 ...$15.00.

perceived latency of an encapsulated component compared to the original version of the component. This is based on data from our field-trial deployment, and in this regard we do benefit from the fact that the encapsulated component is a large, complex, and relatively slow vendor application.

- Usability: We focus on changes to the user experience as well as ease of deployment. Our technique does not substantially alter the user's qualitative experience with the encapsulated component. Deployment is straightforward, as described below.

As alluded to above, we evaluate these points based on data from a field trial with disinterested users at a large aerospace company. Performance measurements were taken from this deployment. We should note that we were not able to control all features of the deployment, and there was some architectural mismatch with respect to communications channels and cloud architecture. Additionally, our system is an operationalized prototype with some inconvenient design choices. The data we obtained nonetheless suggests that our approach offers real benefits – even in the presence of these limitations.

In sections below, we detail the design of the sandbox architecture and the prototype we developed. The principal user-visible change is in the form of window chroming as seen by the user; this is a consequence of expedient features of our prototype design.

We also present two thought experiments to illustrate the potential flexibility our technique provides when designing the attack surface. Finally, we present a structured comparison between our implemented approach and a mainstream approach to highlight the points above regarding the risk product – controlling and minimizing the extent of the attack surface and diminishing the consequences of attack success.

Researchers have been working on encapsulation techniques since at least 1993, when Wahbe[56] introduced the term sandboxing to describe software-based fault isolation (SFI) as a means of preventing distrusted modules from escaping their fault domains. Since then a plethora of sandboxing techniques have been introduced, including mainstream uses such as the Java Runtime Environment in the 1990's and Microsoft's Protected Mode for Internet Explorer 7 in late 2006.

Modern applications that have been sandboxed are typically complex to a degree that defies most existing techniques for finding and fixing vulnerabilities. Sandboxing has helped architects make an application's attack surface smaller and more defensible. Essentially, the architect increases the attacker's costs by applying a security layer to exposed computations. It is particularly valuable for applications that have been compromised repeatedly, are an advantageous target to an attacker, or that cannot be verified using existing methods. In doing so, the defender effectively reduces the verification problem to that of verifying the sandbox.

In practice, sandboxes tend to combine a number of distinct encapsulation techniques to achieve their goal. For example:

- chroot sandboxes (commonly referred to as chroot jails) redirect the root of the filesystem for a specific application [23]. This redirection has the effect of preventing the application from accessing files that are not below what it sees as the filesystem root.

- Google NaCl applies SFI and runtime isolation to prevent memory corruption exploits in native code and to constrain what the native code can do at runtime respectively [57].

- Microsoft Internet Explorer's Protected Mode works by constraining the execution of risky components, such as the HTML render, using rules encoded as integrity levels [12].

- TxBox intercepts all of the system calls made by an application and ensures the calls do not violate a security policy [31].

- TRuE intercepts systems calls, employs SFI, and redirects system resources [31].

The combination of techniques applied by each sandbox varies both the types of policies that can be enforced by the sandbox [19, 26] and how usable the sandbox is for the architect applying it. A chroot jail cannot prevent the execution of arbitrary code via a buffer overflow, but chroot jails can make sensitive files unavailable to the arbitrary code and are quick and easy to deploy when compared to applying Google NaCl to a component. Of course, these two sandboxes are not intended to encapsulate the same types of applications (locally installed desktop applications versus remote web application components), but the comparison does illustrate that the techniques applied by a sandbox have an impact on both how widely a particular sandbox can be applied and how it can fail to protect the underlying system.

Software complexity adds an additional wrinkle. Consider Adobe Reader, which has as a robust PDF parser, a PDF renderer, a JavaScript engine, a Digital Rights Management engine, and other complex components critical to the application's function. It may be extremely difficult to find a sandbox with the right combination of techniques to effectively encapsulate Reader without introducing significant complexity in applying the sandbox itself. Even when breaking up these components for sandboxing purposes, the architect must apply a sandbox where the combination of sandboxing techniques is powerful enough to mitigate the threats faced by the component while also applying the techniques in a manner that is acceptable given the unique characteristics and requirements of each sandboxed computation. Additionally, the sandboxed components must be composed in a way where the composition is still secure [36, 49]. If the combination is not secure, it may be possible for an attacker to bypass the sandbox, e.g. by hopping to an unsandboxed component. The complexity may make the sandbox itself a target, creating an avenue for the attacker to compromise the sandbox directly.

In this paper we propose a sandboxing technique referred to as *in-nimbo sandboxing* to address some of the shortcomings of traditional sandboxing techniques. In-nimbo sandboxing leverages low value computing environments to allow defenders greater control over their attack surface, thus channeling attackers to an attack surface an architect can more confidently defend. The low value environment can be located separately from the target system, so an exploit of the sandboxed component (and any additional sandboxes in the low value environment) has less opportunity to compromise the system we are defending.

Our technique is further motivated in section 2. We discuss an in-nimbo sandbox prototype we built and deployed for Adobe Reader and its performance in section 3. Section 4 compares our in-nimbo sandbox with applying a local sandbox to Reader. We look at thought experiments for applying in-nimbo sandboxing in section 5 before concluding in section 6.

2. IN-NIMBO SANDBOXING

In this section we motivate the need for in-nimbo sandboxing by looking closer at general weaknesses in traditional sandboxes and discuss characteristics of a more suitable environment for executing potentially vulnerable or malicious computations. We then discuss a general model for in-nimbo sandboxing that approximates our ideal environment.

2.1 Why In-Nimbo Sandboxing?

Most mainstream sandboxing techniques are *in-situ*, meaning they impose security policies using only Trusted Computing Bases (TCBs) within the system being defended. In-situ sandboxes are typically retrofitted onto existing software architectures [41, 42, 43, 52, 48, 55, 6] and may be scoped to protect only certain components: those that are believed to be both high-risk and easily isolatable [47, 2]. Existing in-situ sandboxing approaches decrease the risk that a vulnerability will be successfully exploited, because they force the attacker to chain multiple vulnerabilities together [46, 18] or bypass the sandbox. Unfortunately, in practice these techniques still leave a significant attack surface, leading to a number of attacks that succeed in defeating the sandbox. For example, a weakness in Adobe Reader X's sandbox has been leveraged to bypass Data Execution Prevention and Address Space Layout Randomization (ASLR) due to an oversight in the design of the sandbox [20]. Experience suggests that, while in-situ sandboxing techniques can increase the cost of a successful attack, this cost is likely to be accepted by attackers when economic incentives align in favor of perpetrating the attack.[1] The inherent limitation of in-situ techniques is that once the sandbox has been defeated, the attacker is also "in-situ" in the high-value target environment, where he can immediately proceed to achieve his goals.

In order to avoid the inherent limitations of in-situ sandboxing approaches, we propose that improved security may be obtained by isolating vulnerable or malicious computations to *ephemeral computing environments* away from the defended system. Our key insight is that if a vulnerable computation is compromised, the attacker is left in a low-value environment. To achieve his goals, he must still escape the environment, and must do so before the ephemeral environment disappears. The defender controls the means by which the attacker may escape, shaping the overall attack surface to make it more defensible, thereby significantly increasing the cost of attacks compared to in-situ approaches while simultaneously reducing the consequences of successful attacks.

We use the term *ephemeral computing environment* to refer to an ideal environment whose existence is short, isolated (i.e. low coupling with the defended environment), and non-persistent, thus making it fitting for executing even malicious computations. A number of environments may approach the ideal of an ephemeral computing environment, for example, Polaris starts Windows XP applications using an application-specific user account that cannot access most of the system's resources [53]. Occasionally deleting the application's account would further limit the scope of a breach. Terra comes even closer by running application specific virtual machines on separate, tamper resistant hardware [24]. Terra requires custom hardware, complicated virtual machine and attestation architectures, and doesn't outsource risk to third parties. In this paper we focus on cloud environments. Cloud computing closely approximates ephemeral environments, as a virtual computing resource in the cloud is isolated from other resources through the combination of virtualization and the use of separate infrastructure for storage, processing, and communication. It may exist just long enough to perform a computation before all results are discarded at the cloud. We call this approach *in-nimbo sandboxing.*

Executing computations in the cloud gives defenders the ability to customize the computing environment in which the computation takes place, making it more difficult to attack. Since cloud environments are ephemeral, it also becomes more difficult for attackers to achieve persistence in their attacks. Even if persistence is achieved, the cloud computing environment will be minimized with only the data and programs necessary to carry out the required computation, and so will likely be of low value to the attacker.[2] In order to escape to the high-value client environment, the attacker must compromise the channel between the client and the cloud. However, the defender has the flexibility to shape the channel's attack surface to make it more defensible.

To make the idea of in-nimbo sandboxing clear, consider Adobe Reader X. Delegating untrusted computations to the cloud is quite attractive for this application, as Reader in general has been a target for attackers over several years. As described more in section 3, we have built and experimentally deployed in an industrial field trial an in-nimbo sandbox for Reader that sends PDFs opened by the user to a virtual machine running in a cloud. An agent on the virtual machine opens the PDF in Reader. Users interact with the instance of Reader that is displaying their PDF in the cloud via the Remote Desktop Protocol (RDP). When the user is done with the document, it is saved back to the user's machine and the virtual machine running in the cloud is torn down. This example illustrates how a defender can significantly reshape and minimize the attack surface.

2.2 Complementary Prior Work

Martignoni et al. have applied cloud computing to sandbox computations within the cloud in an approach that is complementary to ours [37]. Their trust model reverses ours: whereas our model uses a public, low-trust cloud to carry out risky computations on behalf of a trusted client, they use a

[1] Google Chrome went unexploited at CanSecWest's Pwn2Own contest for three years. Then in 2012, Google put up bounties of $60,000, $40,000, and $20,000 in cash for successful exploits against Chrome. Chrome was successfully exploited three times [25].

[2] There may still be some value to the attacker in the compromised cloud machines, but this is now the cloud provider's problem, which he is paid to manage. This ability to *outsource risk* to the provider is a significant benefit of in-nimbo sandboxing from the point of view of the client.

private, trusted cloud to carry out sensitive computations that the client is is not trusted to perform. They utilize Trusted Platform Modules to attest to the cloud that their client-end terminal is unmodified. They must also isolate the terminal from any malicious computations on the client. Our technique assumes security precautions on the side of the public cloud provider—an assumption we feel is realistic, as cloud providers already have a need to assume this risk.

The scenarios supported by the two techniques are complementary, allowing the application of the two techniques to different components of the same system. For example, Martignoni's sandbox may be used for performing particularly sensitive operations such as online banking, while our technique is useful in the same system for executing untrusted computations from the Internet. These choices reflect the varying trust relationships that are often present in software systems.

2.3 General Model

There are good reasons to reduce the TCB required to execute applications and entire operating systems [38, 40, 39, 50]. The idea is to completely isolate unrelated computations from each other, and to use a TCB that is small enough to be verified, thus reducing and localizing the attack surface to a small, throughly vetted subset of the system's code.

In-nimbo sandboxing addresses this challenge by allowing designers, even when working in legacy environments, flexibility to design TCB(s) to suit their specific context, thus channeling the attacker to a designer-chosen attack surface. This is significantly more flexible as it allows designers to achieve the benefits of a minimal TCB in current commodity hardware/software systems, largely unconstrained by the particular engineering decisions of those systems. When applying in-situ sandboxes, an architect is limited to applying the sandboxing techniques that are supported by the instruction set, operating system, application type, etc., of the system she is trying to defend. These challenges can be particularly difficult to address when vendor software must be sandboxed. However, in-nimbo sandboxes can limit the majority of the attack surface to the communication channel between the client and the cloud. The designer can design a communication channel they are adequately prepared to defend.

In general, in-nimbo sandboxes contain the following:

- A specialized ***transduction mechanism*** in the computing environment we are trying to protect (the principal computing environment) that intercepts invocations of untrusted computations and transfers them to the high value TCB Architecture (see below) on the same system. The transduction mechanism also receives results from the high value TCB architecture and manages their integration into the rest of the system.

- A ***high value TCB architecture*** that sends the untrusted computation and any necessary state to an ephemeral computing asset, separate from the principal computing asset. The high value TCB architecture receives the results of the computation and state from the cloud, verifies both, and transfers them back to the transduction mechanism. We use the term ***TCB architecture*** to reflect the fact that our TCB(s) may be nested in or otherwise cooperate with another TCB (e.g., another sandbox). The nested TCBs can thus compensate for each other's faults and oversights and add redundancy. An attacker must counter each TCB in the architecture to compromise the overall system. In the case of the high value TCB architecture, this could allow an attacker to compromise the system we are trying to defend.

- The cloud executes untrusted computations in a ***low value TCB architecture*** and sends results and state back to the high value TCB architecture.

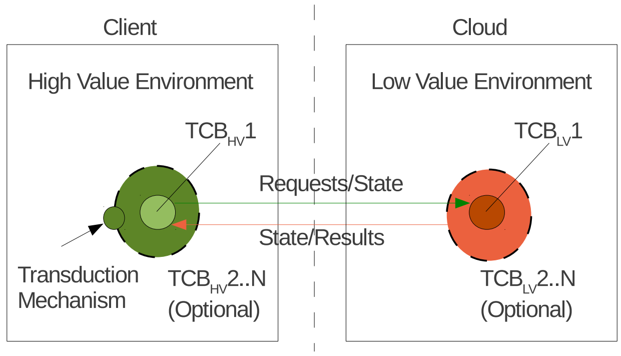


Figure 1: A general model of an in-nimbo sandboxing showing the transduction mechanism and the TCB architecture model. The circles with dashed borders represent one or more TCBs that contain the primary TCBs in each environment. The primary TCB in the high value environment ($\mathbf{TCB_{HV}1}$) is responsible for sending requests and state to the low value environment's primary TCB ($\mathbf{TCB_{LV}1}$). $\mathbf{TCB_{LV}1}$ performs any necessary computations and returns state and results to the $\mathbf{TCB_{HV}1}$, which must verify the results. The transduction mechanism moves computations and results into and out of the high value TCB architecture respectively.

The components and their data flows are depicted in figure 1. By picking these components and the contents of the data flows, the defenders effectively channel an antagonist to an attack surface the defenders are confident they can protect. An attacker must bypass or break every TCB in both TCB architectures or bypass the transduction mechanism to successfully compromise the high value environment.

3. CASE STUDY

In-nimbo sandboxing can be applied to sandboxing entire applications or just selected components/computations. In this section we look at the design of an in-nimbo sandbox for Adobe Reader that we prototyped and experimentally deployed at a large aerospace company. We then discuss the basis comparing our sandbox with an in-situ sandbox for Adobe Reader. In the last section, we discuss other uses for in-nimbo sandboxes.

3.1 Why an In-Nimbo Sandbox for Adobe Reader?

Adobe Reader, hereafter referred to simply as Reader, has been subject to numerous attacks since roughly 2008. To make matters worse, its source code was recently stolen [17]. These attacks are unsurprising since Reader has historically been an obvious path of least resistance for actors seeking to execute targeted attacks [27, 10]:

- It has enjoyed a wide installed base in academia, industry, and government.
- Its purpose is to parse and render potentially extremely complex inputs (PDF files) from potentially unknown sources, where the inputs represent active content created in complex and fully capable programming languages such as JavaScript. Version 1.7 of the PDF Reference [7] is 1,310 pages, while the corresponding ISO standard [30] weighs in at 756 pages in length not counting supplements.[3] To complicate matters, there are numerous variations on the standard meant to accommodate the needs of niches such as archiving [29], graphic exchange [28], and several others. This complexity explodes when we consider all of the standards (e.g., font, cryptography, image, active content, movie, audio, and form standards) on which PDF is dependent.
- PDFs are essential and ubiquitous in many places where layouts must be preserved across platforms and environments.
- PDF viewers, including Reader, tend to be written in unmanaged, weakly typed languages (primarily C/C++) that result in applications that are intractable to verify for security attributes.

These points combine to create a target for attack that is likely continue to offer vulnerabilities and is likely available on a multitude of machines worth compromising.

One common suggestion from the security community is to deal with this issue by using alternative PDF viewers [8, 15, 32], which may not suffer from the same vulnerabilities. But in fact many of these share code, because Adobe licenses its PDF parser/renderer [14]. The advice nonetheless has some merit because even if an alternative viewer contains the exact same vulnerability as Reader, an exploit PDF that targets Reader won't necessarily work "out of the box" on an alternative. Many organizations, however, have grown dependent on PDF's more exotic features, such as fly-through 3D models, forms that can be filled out and then digitally signed by a user using hardware tokens, or embedded multimedia in a wide range of formats. At the time of this writing (late-2013), many of these types of features are not supported by many of the alternative viewers.

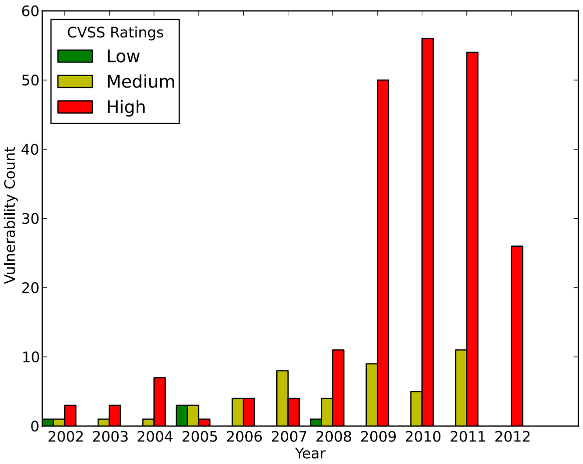


Figure 2: The distribution of vulnerabilities discovered in Reader where data was available for the entire year.

Let us consider, therefore, an organization with needs that require use of Reader, which has a significant, complex attack surface to defend. We examined the vulnerability streams (essentially RSS feeds of vulnerability reports) provided by NIST's National Vulnerability Database. We only used streams that contained a full years worth of data. The census results in figure 2 show an increasing number of vulnerabilities per year since 2008,[4] with a possible downward trend starting in 2011. There is great diversity in where the vulnerabilities reside in the code, as shown in figure 3. (Sometimes a single vulnerability entry describes multiple vulnerabilities,[5] thus our results undercount how many vulnerabilities have actually been reported.) The vulnerable component distribution in figure 3 was determined by coding all entries by hand that were identified as belonging to Reader and where the coded Common Weakness Enumeration (CWE) implicitly identified a component or a component was explicitly named. As a result, our results likely understate the diversity of vulnerabilities in Reader.

Adobe attempted to address these issues in mid-2010 by sandboxing a subset of Adobe Reader X, and this could be a cause of the decline shown in figure 2. (It is not clear if the apparent slight decrease in reported Reader vulnerabilities is attributable to better application security practices at Adobe, the application of a sandbox, more bundled reports, or some other cause.) As mentioned in section 2, cracks in this sandbox have now been publicly exposed. While the number of vulnerabilities appears to have decreased starting in 2011, many of the vulnerabilities did became more difficult to exploit due to the sandbox.

Others have attempted to detect malicious PDFs [9, 22, 33, 54, 35, 45, 51, 21, 5] with modest success even in the best cases. A systematic analysis [34] substantiates the inadequacy of many techniques for detecting malicious PDFs. We believe that even if detection methods advance, they will never be adequate against advanced attackers. The sheer

[3] Adobe's PDF Reference looks at the file format using Reader's implementation as the point of reference, while the standard just looks at the format on its own without considering a concrete implementation.

[4] The number of vulnerabilities was determined by counting how many vulnerability entries contained a `cpe-lang:fact-ref` field with a name attribute containing the text `adobe:reader`, `adobe:acrobat`, or `adobe:acrobat_reader`.

[5] When a vendor tracks multiple vulnerabilities separately internally but discloses them in one bulletin with too few details to separate them aside from proprietary vulnerability identifiers, NIST tends to report all of the issues under one Common Vulnerability Enumeration identifier.

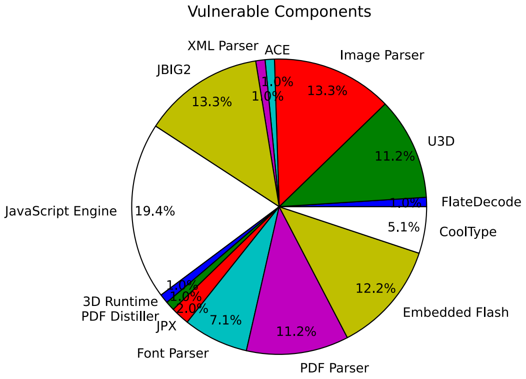


Figure 3: The distribution of vulnerabilities in Reader amongst identifiable components. This distribution is intended to show that problems in Reader are so diverse there is no clear place to concentrate defensive efforts outside of sandboxing.

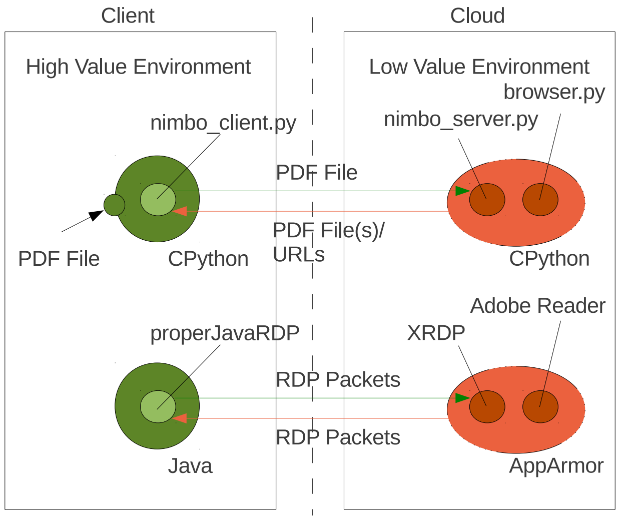


Figure 4: An in-nimbo sandbox for Adobe Reader.

complexity involved in detecting every possible attack in a format as massive as PDF is prohibitive. Additionally, several of Reader's components that have been vulnerable take inputs that involve fully capable programming languages such as JavaScript.

Additional comparison between in-nimbo sandboxing for PDF's and other defensive techniques appears in the appendix. The appendix characterizes several other defensive techniques, defines the criteria used to compare them, and compares the approaches in a criteria matrix.

In-nimbo sandboxing allows Reader to be executed in a low value environment, thus bypassing the complexity concerns detailed in this section in dealing with the richness of the PDF standard. This analysis led us to design and build an in-nimbo sandbox for Adobe Reader.

3.2 Design

To demonstrate the ability of an in-nimbo sandbox to support rich features, we set out with the goal of designing and building a sandbox for Reader that can support a user clicking links in a PDF, filling out and saving forms, interacting with multiple PDFs in the same session, printing PDFs, copying and pasting data to and from a PDF, and interacting with an embedded 3D model. We neglect features such as signing PDF documents with a smart card and several other non-trivial features Adobe advertises (though these features are likely supported via the RDP implementation we used). As an additional convenience, we decided that any Reader settings changed by the user should persist across their sandbox sessions. These design choices ensure the user's interactions with in-nimbo Reader are substantially similar to their typical interactions with in-situ Reader. In fact, aside from a few missing but less commonly used features, the user's interaction only differs in the appearance of the window chroming.

Figure 4 shows a high-level structural model of our prototype in-nimbo sandbox for Adobe Reader. The transduction mechanism consists of a file association between the PDF file extension and nimbo_client.py. This small script and its infrastructure is the primary TCB in the high value environment (24 lines of code). When a PDF file is opened nimbo_client.py transfers the file to nimbo_server.py running in a cloud virtual machine instance. If a sandboxed session does not yet exist, a new session is created. Otherwise, the new PDF is opened as a new tab in the already open instance of Reader.

The user interacts with PDFs opened in the sandboxed version of Reader over an encrypted RDP connection. When Reader, the RDP session, or the RDP client is closed, all of the PDFs in the sandbox are sent back to nimbo_client.py. The PDFs must be returned to the high value client environment because the user may have performed an operation in the sandbox that changed the PDF, such as filling out and saving a form. After the PDFs are returned, nimbo_server.py restores the home directory of the in-nimbo user account that runs Reader to its original state. The cloud environment can always start virtual machines from a clean template, but alternatively resetting the home directory can enable a virtual machine to be re-used, e.g., due to high congestion. The account that runs Reader has limited access to the rest of the system.

When a user clicks a web link in a PDF, that link is sent to the nimbo_client.py and opened on the workstation. If the user does not want links to be opened on their workstation due to the risk of loading a potentially malicious site, they could instead have links opened in an in-nimbo version of their favorite browser. In this way, in-nimbo sandboxes can be composed. Sandbox composition is useful in this case because it prevents any one sandbox from becoming too complicated, and, in particular, having an overly rich attack surface.

3.3 Performance

Our prototype sandbox's performance evaluation is limited by two factors: transfer rates and inefficiencies in the cloud virtual machine set-up for the field trial. But even with inefficiencies in our virtual machine set-up, user perception of performance is comparable with Reader's performance locally. Transfer rates dominate the time between when a user opens a PDF and when the user can interact with the PDF, but this rate is typically limited by the connection's upload rate at the workstation. As a result, the upload time statistics presented in this section are not intrinsic to in-nimbo sandboxing and may vary with different connections. The statistics were gathered in our field trial using an in-nimbo sandbox that was deployed as the company would typically deploy cloud applications. The measurements as a whole provide evidence that the approach performs acceptably for typical users.

The Python scripts use the same efficient file transfer schemes used by FTP clients. The server prototype in the cloud implements several optimizations, including queuing virtual machines with Reader already running and waiting for a user to connect. However, the infrastructure software running on the cloud virtual machines is not optimized for our use case. For example, establishing a connection with RDP would be faster if the connection was initialized ahead of time (i.e. subsequent connections via RDP to the virtual machine are much faster than the first connection). This is a side-effect of our choice to use Linux, X server, and xrdp. The issue does not exist on Windows with the RDP server Microsoft provides. It is also possible that xrdp can be modified to remove the issue. Table 1 summarizes results from our industrial collaborator who used an internal cloud and a Microsoft Windows client.

We measured performance with our collaborator by opening a 1 megabyte (MB) PDF ten times. We decided to use a 1 MB PDF after inspecting a corpus of about 200 PDFs characteristic of the use cases of the users and averaging the sizes of its contents. The corpus was collected by an engineer over three years and included the types of PDFs an average engineer encounters throughout their day: brochures, technical reports, manuals, etc. Most PDFs in the corpus were on the order of a few hundred kilobytes, but a small number of the PDFs were tens of MB in size.

For our measurements the high-value environment was 1,800 miles away from the datacenter hosting our cloud. While we parallelize many of the operations required to get to the point where the user can interact with the PDF, the largest unit of work that cannot be comfortably parallelized is transferring the PDF itself. In our tests, the user could interact with our 1 MB PDF within 2.1 seconds, which compares favorably to the 1.5 seconds it takes to interact with a PDF run in Reader locally instead of in-nimbo. The Reader start-up difference is due to the fact that the virtual machine is much lighter than the workstation. The virtual machine doesn't need to run anti-virus, firewalls, intrusion prevention systems, productivity software, and other applications that slow down the workstation.

Though our sandbox increases the startup time, sometimes by several seconds in the case of large PDFs due to the transfer time, we observed no performance issues on standard broadband connections in the United States when interacting with the PDF. The sandbox also performed well when running malicious PDFs that were collected when they were used in attempted attacks targeted at our collaborator's employees. The malware did not escape the sandbox, nor did it persist across sessions. Our results suggest that our technique is currently best applied to longer running, interactive computations unless the running time for the entire initialization process is negligible. The aerospace company we worked with is currently evaluating whether or not to transition the sandbox into production for day-to-day use by high-value targets within the company (e.g. senior executives).

PDF Size	1 MB
Average upload time	2.1 +/- 0.3 seconds*
Average Adobe Reader start time in-nimbo	0.5 seconds
Average time to establish RDP channel	1.5 seconds
Average time until user can interact	2.1 seconds
Distance from client to cloud	1,800 miles
Average Adobe Reader start time in-situ	1.5 seconds

Table 1: Performance metrics for an in-nimbo sandbox using a cloud internal to an enterprise and a Microsoft Windows client. Figures based on 10 runs. The upload time is the primary bottleneck. *Confidence level: 95%

3.4 Limitations

The sandbox prototype does not support the most recent features of PDF because we used the Linux version of Reader, which is stuck at version 9. This limitation is an accidental consequence of our expedient implementation choices and is not intrinsic to in-nimbo sandboxing. It is possible, for example, to instead run Windows with the latest version of Reader in the cloud virtual machine, but this setup would not substantially influence our performance results given the dominance of the transfer rate. (Furthermore, it is possible to run newer Windows versions of Reader in Linux. Adobe Reader X currently has a Gold rating in the WINE AppDB for the latest version of WINE [1]. The AppDB claims that a Gold rating means the application works flawlessly after applying special, application specific configuration to WINE.)

Our malware test of the sandbox is limited by the fact that we didn't have access to malicious PDFs directly targeting Linux or malware that would attempt to break out of our sandbox.

4. IN-NIMBO ADOBE READER VS. IN-SITU ADOBE READER X

In this section we make a structured comparison between our in-nimbo sandbox for Reader with an in-situ Adobe Reader. First, we summarize the framework we'll use for the comparison, and then we apply the framework. The purpose of the framework is to support a systematic exploration of our hypothesis that in-nimbo sandboxing leads to attack surfaces that (1) are smaller and more defensible and (2) offer reduced consequences when successful attacks do occur. The framework is necessarily multi-factorial and qualitative because quantification of attack surfaces and the potential extent of consequences remains elusive.

4.1 Structuring the Comparison

To compare sandboxes we consider what happens when the sandbox holds, is bypassed, or fails. A sandbox is *bypassed* when an attacker can accomplish his goals by jumping from a sandboxed component to an unsandboxed component. A sandbox *fails* when an attacker can accomplish his goals from the encapsulated component by directly attacking the sandbox. The key distinction between a sandbox bypass and a failure is that any malicious actions in the case of a bypass occur in a component that may have never been constrained to prevent any resulting damage. In a failure scenario, the malicious actions appear to originate from the sandbox itself or the encapsulated component, which creates more detectable noise than the case of a bypass. A bypass can occur when an insufficient security policy is imposed, but a failure requires a software vulnerability. These dimensions help us reason about the consequences of a sandbox break, thus allowing us to argue where in the *consequences* spectrum a particular sandbox falls within a standard risk matrix. To place the sandbox in a risk matrix's *likelihood* spectrum (i.e. the probability of a successful attack given the sandbox), we consider how "verifiable" the sandbox is. Our risk matrix only contains categories (e.g. low, medium, or high) that are meaningful to the comparison at hand. Finally, we rank the outcomes that were enumerated within the argument by their potential hazards, which helps highlight the difference between risk categories.

4.2 Comparing In-Nimbo Adobe Reader to In-Situ Adobe Reader

Adobe Reader X's sandbox applies features of Microsoft Windows to limit the damage that an exploit can do. The Reader application is separated into a low privilege (sandboxed) principal responsible for parsing and rendering PDFs and a user principal responsible for implementing higher privilege services such as writing to the file system. The sandboxed principal is constrained using limited security identifiers, restricted job objects, and a low integrity level. The higher privilege principal is a more stringently vetted proxy to privileged operations. The sandboxed principal can interact with the user principal over a shared-memory channel. The user principal enforces a whitelist-based security policy on any interactions from the sandboxed principal that the system administrator can enhance. Under ideal circumstances the sandboxed principal is still capable of reading secured objects (e.g., files and registry entries),[6] accessing the network, and reading and writing to the clipboard without help from the user principle.

The consequences of an attack on Reader, even when the sandbox holds, are high. In its default state, a PDF-based exploit could still exfiltrate targeted files over the network without any resistance from the sandbox. If the sandbox is successfully bypassed, the attacker can leverage information leakages to also bypass mitigations such as ASLR as mentioned in section 2. Such bypasses, which are publicly documented, are likely to be serious enough to allow a successful attack to install malware on the targeted machine. If other bypass techniques exist, they could allow an attacker to perform any computations the user principal can perform. These outcomes are ranked from most to least damaging in figure 5. Overall, the consequences generally fall into one of the following categories:

- The integrity of the PDF and viewer are compromised
- The confidentiality of the PDF is compromised
- The availability of reader is compromised
- The security (confidentiality, integrity, availability) of the cloud infrastructure is compromised
- The security of the high value environment is compromised

In-Situ Reader X
Install malware on the defended workstation
Perform any computation the user principal can perform
Exfiltrate workstation data on the network
Read files on the workstation filesystem

In-Nimbo Reader
Spy on opened PDFs in the cloud
Abuse cloud resources for other computations

Figure 5: Likely sandbox "consequence outcomes" ranked from most damaging at the top to least damaging at the bottom. Each sandbox's outcomes are independent of the other sandbox's.

The Reader sandbox is moderately verifiable. It is written in tens of thousand of lines of C that are heavily based on the open-source sandbox created for Google Chrome. The design and source code were manually evaluated by experts from several independent organizations who also implemented a testing regime. According to Adobe, source code analysis increased their confidence in the sandbox as its code was written. The operating system features on which it depends are complex; however, they were implemented by a large software vendor that is known to make use of an extensive Secure Development Lifecycle for developing software [44].

Figure 6 summarizes our qualitatively defined risk for Reader X's sandbox against Reader running in our in-nimbo sandbox. The in-nimbo sandbox has a lower consequence (from the standpoint of the user) because exploits that are successful against Reader may only abuse the cloud instance Reader runs in. The operator of the cloud instance may re-image the instance and take action to prevent further abuse. However, to abuse the cloud instance the attacker would have to both successfully exploit Reader and bypass or break additional sandboxing techniques we apply in the cloud. The exploit must either not crash Reader, or its payload must survive the filesystem restoration and account log-off that would occur if Reader crashed due to the exploit (see 3.1 for details).

The attacker could potentially *bypass* the sandbox by tricking our mechanism into opening the PDF in a different locally installed application capable of rendering PDFs. For example, the attacker may disguise the PDF file as an HTML file, causing it to be opened in the browser if the transduction mechanism is only based on file associations. The browser might have an add-on installed that inspects the document, determines it is a PDF regardless of extension, and then renders the PDF. While this attack would eliminate the benefit of the sandbox, it is not likely to be successful if the user verifies there is not a local PDF viewer installed/enabled (an important part of configuration). The transduction mechanism can also simply use a richer means of determining whether or not a file is a PDF.

The sandbox could *fail* in a way that compromises the user by either an exploitable vulnerability in our 273 line Python-

[6] Windows Integrity Levels can prevent read up, which stops a process from reading objects with a higher integrity level than the process. Adobe did not exercise this capability when sandboxing Adobe Reader X.

based TCB (and its infrastructure), the Java RDP client we use, or a kernel mode vulnerability exploitable from either. Such a failure would require first successfully compromising the cloud instance as discussed earlier and then finding an interesting vulnerability in our small, typesafe components. In other words, a failure of the sandbox requires that the TCBs in both the client and the cloud fail.

Another potential point of concern is that the cloud instance's hypervisor could be compromised, thus compromising other virtual machines managed by that hypervisor or even the entire cloud. We do not consider this issue in our analysis because our sandbox is a use of the cloud, not an implementation of a cloud. One of the key selling points behind using a cloud environment is that the provider manages the infrastructure; they take on the risk and management expenses. The ability to outsource risk that cannot be eliminated to a party that is willing to assume the risk is a key advantage of our approach. Additionally, our technique does not add a new threat to clouds in the sense that anyone can rent access to any public cloud for a small sum of money and attempt to compromise the hypervisor. Finally, we are primarily sandboxing computations because we don't trust them. In the case of the Reader sandbox, a compromise could cause sensitive PDFs to be stolen, which would still be better than the compromise of an entire workstation full of sensitive information.

In short, the in-nimbo sandbox is easier to verify and requires more work for an attacker to achieve enough access to compromise the client. Adobe Reader X's sandbox is harder to verify and allows the workstation it is running on to be compromised even if the sandbox holds. Due to the well known characteristics of each sandbox, we consider this evaluation to be reasonable evidence of the validity of our hypotheses that in-nimbo sandboxing leads to smaller, more defensible attack surfaces and reduced consequences in the event of successful attacks. While we only evaluated one sandbox in addition to our in-nimbo sandbox and for only one application, the results of the evaluation are largely influenced by issues that are fundamental to in-situ sandboxing when compared to in-nimbo sandboxing.

Consequence		
High		In-Situ Reader X
Low	In-Nimbo Reader	
	Easy	*Moderate*

Likelihood (Verifiability)

Figure 6: A grid summarizing our evaluation of Reader X and our In-Nimbo sandbox for Reader.

5. IN-NIMBO THOUGHT EXPERIMENTS

In this section we consider the potential of applying the in-nimbo sandboxing technique for a subset of HTML5 and for protecting proprietary data. There are other examples (not elaborated here) that would be similar to our sandbox for Reader, such as an in-nimbo sandbox for Microsoft Word or Outlook.

5.1 Selective Sandboxing

HTML5 specifies a canvas element [13] that provides scripts with a raster-based surface for dynamically generating graphics. By default, browsers must serialize the image to a PNG for presentation to the user (other raster formats may be specified). For an animated surface, the serialized image represents the currently visible frame. Unfortunately, early implementations of such rich browser features have continually been prone to vulnerabilities. CVE-2010-3019 documents a heap-based buffer overflow in Opera 10's implementation of HTML5 canvas transformations [3]. A stack-based overflow was also recently discovered in Mozilla Firefox's HTML5 canvas implementation[4]. As a result, a risk-focused user may desire that HTML5 canvas computations be sandboxed.

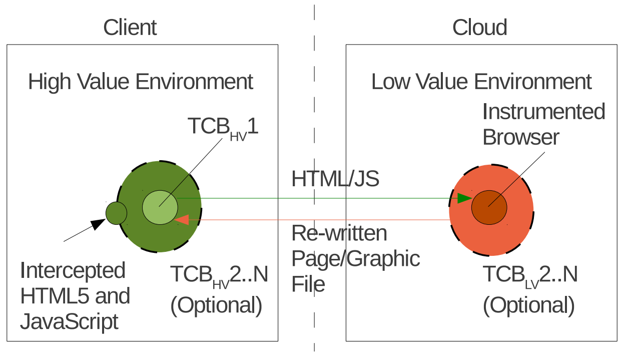

Client
Cloud
High Value Environment
Low Value Environment
TCB$_{HV}$1
Instrumented Browser
HTML/JS
Re-written Page/Graphic File
Intercepted HTML5 and JavaScript
TCB$_{HV}$2..N (Optional)
TCB$_{LV}$2..N (Optional)

Figure 7: The model for an in-nimbo sandbox for HTML5 canvases.

Figure 7 shows the model of an HTML5 canvas in-nimbo sandbox. In this case, the transduction mechanism is a proxy running between a browser and the Internet. The transduction mechanism intercepts and inspects HTML pages for canvas declarations. When a canvas declaration is detected, the proxy collects all referenced JavaScript code and sends the page and scripts to the high value TCB architecture (the client). The client sends the collected HTML and JavaScript to the cloud instance, which utilizes an instrumented browser to render any drawing on the canvas. The canvas is replaced with the image file the browser generates (per the HTML5 standard) when the rendering script finishes. When a loop in the rendering script is detected (i.e. an animation is present), the canvas declaration is replaced with an image tag pointing to a 3 second animated GIF composed of all of the frames the script rendered in that time period. All JavaScript that drew on the canvas is removed, and the cloud returns the re-written JavaScript, HTML, and PNG to the client. The client verifies the image file and checks that no unexpected code/markup changes have been made before sending the results back to the proxy. The proxy passes the modified results to the browser.

This sandbox would effectively confine exploits on HTML5 canvas implementations to our low value computing environment. Furthermore, it would reduce the verification problem from verifying the full HTML5 canvas implementation and its dependencies to that of verifying raster image formats supported by the canvas tag and ensuring that no code has been added to the intercepted files (i.e., code has only been removed and/or canvas tags have been replaced with image tags). While the sandbox does not support animated canvases longer than 3 seconds or whose visual representation is dependent on real-time user input, a user who cannot accept such limitations can use a browser that is fully sandboxed in-nimbo such as Reader was in the previous section. It is also possible that an alternative implementation of this sandbox could support longer animations and user input.

5.2 Protecting Proprietary Algorithms

Modern audio players allow users to manage libraries of tens of thousands of songs and automatically perform experience-enhancing operations such as fetching album covers from the Internet. More advanced players also attempt to identify songs and automatically add or fix any missing or corrupt metadata, a process known as auto-tagging. Unfortunately, the act of robustly parsing tags to identify an album and artist to fetch a cover is potentially error prone and complicated. CVE-2011-2949 and CVE-2010-2937 document samples of ID3 parsing vulnerabilities in two popular media players [16, 11]. Furthermore, an audio player vendor may consider all steps of the waveform analysis they use to identify untagged audio files to be proprietary. To address these concerns, the vendor may wish to design their application to make use of an in-nimbo sandbox to perform these operations.

Figure 8 shows the model of a possible in-nimbo sandbox for fetching album covers and performing auto-tagging. The transduction mechanism is the audio player itself. When the player detects an audio file that is new it sends the file to the high value TCB architecture (the client). The client sends the audio file to the cloud instance, which performs the task of automatically adding any missing tags to the audio file and fetching the correct album cover. The cloud sends the tagged audio file and album cover to the client, where it will be verified that only the audio files tags have changed, that they comply with the correct tagging standard, and that the album cover is of the expected format and well formed. The client will then send the verified audio file and album cover to the audio player, which will place both into their correct places in the library.

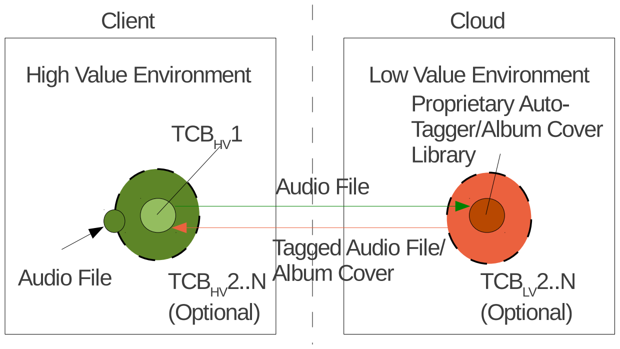

Client
Cloud
High Value Environment
Low Value Environment
Proprietary Auto-Tagger/Album Cover Library
TCB$_{HV}$1
Audio File
Tagged Audio File/ Album Cover
Audio File
TCB$_{HV}$2..N (Optional)
TCB$_{LV}$2..N (Optional)

Figure 8: The model for an in-nimbo sandbox for an auto-tagging audio player.

Assuming that all managed audio files make their way through the in-nimbo sandbox at least once, this sandbox effectively mitigates the risk of robustly parsing tags while also not exposing the inner-workings of the proprietary waveform algorithm. Potential limitations are curtailed by intentionally designing the application to utilize an in-nimbo sandbox.

6. CONCLUSIONS AND FUTURE WORK

In this paper we argued that we can improve system security (confidentiality, integrity, availability) by moving untrusted computations away from environments we want to defend. We did so by first introducing one approach for achieving that idea, a category of sandboxing techniques we refer to as in-nimbo sandboxing. In-nimbo sandboxes leverage cloud computing environments to perform potentially vulnerable or malicious computations away from the environment that is being defended. Cloud computing environments have the benefit of being approximately ephemeral, thus malicious outcomes do not persist across sandboxing sessions. We believe this class of sandboxing techniques is valuable in a number of cases where classic, in-situ sandboxes do not yet adequately isolate a computation.

We argued that in-situ sandboxing does not adequately reduce risk for Adobe Reader, thus motivating us to build an in-nimbo sandbox for Reader. We then discussed the design of an in-nimbo sandbox for Reader and presented a structural argument based on five evaluation criteria that suggests that it is more secure and that, with respect to performance, has a user experience latency subjectively similar to that of Reader when run locally. After arguing that our sandbox is more secure than an in-situ sandbox for Reader, we looked at how in-nimbo sandboxes might be built for a couple of other applications that represent different in-nimbo use cases.

Our argument for why our sandbox is better is structured but necessarily qualitative. We believe that many security dimensions cannot now be feasibly quantified. We nonetheless suggest that structured criteria-based reasoning building on familiar security-focused risk calculus can lead to solid conclusions. Indeed, we feel the approach is an important intermediate step towards the ideal of quantified and proof-based approaches.

7. ACKNOWLEDGEMENT

This material is based upon work supported by the Army Research Office under Award No. W911NF-09-1-0273 and by the Air Force Research Laboratory under Award No. FA87501220139.

8. REFERENCES

[1] AppDB Adobe Reader. http://goo.gl/Fx9pd.
[2] Chromium sandbox. http://www.chromium.org/developers/design-documents/sandbox/.
[3] National vulnerability database (NVD) national vulnerability database (CVE-2010-3019). http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3019.
[4] National vulnerability database (NVD) national vulnerability database (CVE-2013-0768). http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0768.
[5] PDF x-RAY. https://github.com/9b/pdfxray_public.
[6] What is protected view? - word - office.com. http://office.microsoft.com/en-us/word-help/what-is-protected-view-HA010355931.aspx.
[7] *PDF Reference*, sixth edition ed. Adobe Systems Incorporated, Nov. 2006.
[8] Two new vulnerabilities in Adobe Acrobat Reader. http://www.f-secure.com/weblog/archives/00001671.html, Apr. 2009.
[9] Anatomy of a malicious PDF file. http://goo.gl/VlLmU, Feb. 2010.

[10] Military targets. http://www.f-secure.com/weblog/archives/00002203.html, July 2011.

[11] National vulnerability database (NVD) national vulnerability database (CVE-2011-2949). http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-2949, Oct. 2011.

[12] Understanding and working in protected mode internet explorer. http://msdn.microsoft.com/en-us/library/bb250462(v=vs.85).aspx, Feb. 2011.

[13] 4.8.11 the canvas element - HTML5. http://www.w3.org/TR/html5/the-canvas-element.html#the-canvas-element, Mar. 2012.

[14] Adobe PDF library SDK | adobe developer connection. http://www.adobe.com/devnet/pdf/library.html, Aug. 2012.

[15] Google warns of using adobe reader - particularly on linux. http://www.h-online.com/security/news/item/Google-warns-of-using-Adobe-Reader-particularly-on-Linux-1668153.html, Aug. 2012.

[16] National vulnerability database (NVD) national vulnerability database (CVE-2010-2937). http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-2937, Jan. 2012.

[17] Arkin, Brad. Illegal access to Adobe source code. http://blogs.adobe.com/asset/2013/10/illegal-access-to-adobe-source-code.html, Oct. 2013.

[18] Buchanan, K., Evans, C., Reis, C., and Sepez, T. Chromium blog: A tale of two pwnies (Part 2). http://blog.chromium.org/2012/06/tale-of-two-pwnies-part-2.html, June 2012.

[19] Clarkson, M. R., and Schneider, F. B. Hyperproperties. *Journal of Computer Security 18*, 6 (2010), 1157–1210.

[20] Delugre, G. Bypassing ASLR and DEP on Adobe Reader X - Sogeti ESEC Lab. http://esec-lab.sogeti.com/post/Bypassing-ASLR-and-DEP-on-Adobe-Reader-X, June 2012.

[21] Esparza, J. peepdf - PDF analysis and creation/modification tool. http://code.google.com/p/peepdf/.

[22] Fratantonio, Y., Kruegel, C., and Vigna, G. Shellzer: a tool for the dynamic analysis of malicious shellcode. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2011), RAID'11, Springer-Verlag, pp. 61–80.

[23] Friedl, S. Best practices for UNIX chroot() operations. http://www.unixwiz.net/techtips/chroot-practices.html.

[24] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 193–206.

[25] Goodin, D. At hacking contest, Google Chrome falls to third zero-day attack (Updated). http://arstechnica.com/business/news/2012/03/googles-chrome-browser-on-friday.ars, Mar. 2012.

[26] Hamlen, K. W., Morrisett, G., and Schneider, F. B. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS) 28*, 1 (2006), 175–205.

[27] Higgins, K. Spear-phishing attacks out of China targeted source code, intellectual property. http://goo.gl/8RzyT, Jan. 2010.

[28] International Organization for Standardization, I. ISO 15929:2002 - International Organization for Standardization. http://goo.gl/SUP1A.

[29] International Organization for Standardization, I. ISO 19005-2:2011 - International Organization for Standardization. http://goo.gl/mtHWw.

[30] International Organization for Standardization, I. ISO 32000-1:2008 - International Organization for Standardization.

[31] Jana, S., Porter, D. E., and Shmatikov, V. TxBox: Building secure, efficient sandboxes with system transactions. In *2011 IEEE Symposium on Security and Privacy (SP)* (May 2011), IEEE, pp. 329–344.

[32] Landesman, M. Free PDF readers: Alternatives to Adobe Reader and Acrobat. http://antivirus.about.com/od/securitytips/tp/Free-Pdf-Readers-Alternatives-To-Adobe-Reader-Acrobat.htm.

[33] Laskov, P., and Srndic, N. Static detection of malicious JavaScript-bearing PDF documents. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, ACM, pp. 373–382.

[34] Maiorca, D., Corona, I., and Giacinto, G. Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious PDF files detection. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 119–130.

[35] Maiorca, D., Giacinto, G., and Corona, I. A pattern recognition system for malicious PDF files detection. In *Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition* (Berlin, Heidelberg, 2012), MLDM'12, Springer-Verlag, pp. 510–524.

[36] Mantel, H. On the composition of secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy, 2002* (2002), IEEE, pp. 88–101.

[37] Martignoni, L., Poosankam, P., Zaharia, M., Han, J., McCamant, S., Song, D., Paxson, V., Perrig, A., Shenker, S., and Stoica, I. Cloud Terminal: Secure access to sensitive applications from untrusted systems. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 14–14.

[38] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy (SP), 2010* (2010), IEEE, pp. 143–158.

[39] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Seshadri, A. How low can you go?: Recommendations for hardware-supported minimal TCB code execution. *SIGOPS Oper. Syst. Rev. 42*, 2 (Mar. 2008), 14–25.

[40] McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev. 42*, 4 (Apr. 2008), 315–328.

[41] McQuarrie, L., Mehra, A., Mishra, S., Randolph, K., and Rogers, B. Inside Adobe Reader Protected Mode - part 1 - design. http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html, Oct. 2010.

[42] McQuarrie, L., Mehra, A., Mishra, S., Randolph, K., and Rogers, B. Inside adobe reader protected mode - part 2 - the sandbox process. http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-%E2%80%93-part-2-%E2%80%93-the-sandbox-process.html, Oct. 2010.

[43] McQuarrie, L., Mehra, A., Mishra, S., Randolph, K., and Rogers, B. Inside adobe reader protected mode - part 3 - broker process, policies, and inter-process communication. http://blogs.adobe.com/asset/2010/11/inside-adobe-reader-protected-mode-part-3-broker-process-policies-and-inter-process-communication.html, Nov. 2010.

[44] Michael Howard, and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, May 2006.

[45] Nedim Srndic, and Pavel Laskov. Detection of malicious PDF files based on hierarchical document structure. In *Network and Distributed System Security Symposium* (2013).

[46] Obes, J., and Schuh, J. Chromium blog: A tale of two pwnies (Part 1). http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html, May 2012.

[47] Sabanal, P., and Yason, M. Playing in the Reader X sandbox. *Black Hat USA Briefings* (July 2011).

[48] Schuh, J. Chromium blog: The road to safer, more stable, and flashier flash. http://blog.chromium.org/2012/08/the-road-to-safer-more-stable-and.html, Aug. 2012.

[49] Sewell, P., and Vitek, J. Secure composition of insecure components. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop, 1999.* (1999), IEEE, pp. 136–150.

[50] Singaravelu, L., Pu, C., Hartig, H., and Helmuth, C. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev. 40*, 4 (Apr. 2006), 161–174.

[51] Smutz, C., and Stavrou, A. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 239–248.

[52] Stender, S. Inside adobe reader protected mode - part 4 - the challenge of sandboxing. http://blogs.adobe.com/asset/2010/11/inside-adobe-reader-protected-mode-part-4-the-challenge-of-sandboxing.html, Nov. 2010.

[53] Stiegler, M., Karp, A. H., Yee, K.-P., Close, T., and Miller, M. S. Polaris: Virus-safe computing for windows XP. *Commun. ACM 49*, 9 (Sept. 2006), 83–88.

[54] Tzermias, Z., Sykiotakis, G., Polychronakis, M., and Markatos, E. P. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security* (New York, NY, USA, 2011), EUROSEC '11, ACM, pp. 4:1–4:6.

[55] Uhley, P., and Gwalani, R. Inside flash player protected mode for firefox. http://blogs.adobe.com/asset/2012/06/inside-flash-player-protected-mode-for-firefox.html, June 2012.

[56] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev. 27*, 5 (Dec. 1993), 203–216.

[57] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM 53*, 1 (Jan. 2010), 91–99.

APPENDIX

A. PDF DEFENSE CONCEPT RANKINGS

Alternative mitigations for PDF. Other possible mitigations, such as transforming a PDF to a more defensible format, are important to consider. As noted, these must contend with the challenges of supporting some of the exotic features mentioned in section 3.1 if they are to be usable by the widest possible audience. Table 2 summarizes the main criteria that we believe a more optimal defense concept for PDF must manifest. Table 3 shows a criteria matrix with some evaluation points to assist in comparing format transformation to an in-nimbo sandbox. We compared more than two approaches for defending PDF, but this is sufficient to illustrate the criteria-based approach.

Table 2: A summary of the criteria a more optimal PDF defense might manifest.

Criteria	Description
Simple File Transduction	The PDF must be able to cross between a native and either transformed or sandboxed contexts. For example, in local sandboxing a PDF must be intercepted and placed in the sandbox's file system. There must be a simple and efficient mechanism for performing this operation.
Native State Persistence	Modern readers may persist state in the form of the last page viewed, digital signatures on subsets of the file, and form content. This state should be maintained as it would be if the PDF was interacted with in a traditionally installed fully featured reader.
Advanced PDF Feature Support	PDFs can be used to interchange Flash content, movies, audio, and 3D models to name few examples of advanced content. Additionally, segments of a PDF can be digitally signed, locked for printing, etc. Maintaining support for many of these features (not necessarily all) is paramount.
Low Breakout Risk	There should be little chance that an exploit succeeds in compromising a target workstation.
High Breakout Recovery	If an exploit breaks the defense concept, recovery should be trivial.
Low Performance Overhead	The defense concept should not unacceptably harm performance.
Low Adoption Overhead	Defense concepts should be easy for individuals and enterprises to effectively place into operation.

Table 3: A criteria matrix that compares Format Transformation and In-Nimbo Sandboxing in defending PDF to an unsandboxed version of Reader.

	Format Trans.	In-Nimbo Sandbox
Simple File Transduction	Minimal Change	Degraded
Native State Persistence	Much Degraded	Improved
Adv. PDF Support	Much Degraded	Minimal Change
Low Breakout Risk	Improved	Much Improved
High Breakout Recovery	Minimal Change	Improved
Printing Support	Minimal Change	Minimal Change
Low Performance Ovrhd	Minimal Change	Minimal Change
Low Adoption Ovrhd	Minimal Change	Minimal Change

To appear in the 11th International Conference on Information Technology : New Generations (ITNG 2014), Special track on: MDCBSE: Model-Driven, Component-Based Software Engineering, Las Vegas, NV, 7-9 April 2014.

Model-based Assistance for Making Time/Fidelity Trade-offs in Component Compositions

Vishal Dwivedi, David Garlan, Jürgen Pfeffer and Bradley Schmerl
Institute for Software Research
Carnegie Mellon University, Pittsburgh, PA
{vdwivedi, garlan, jpfeffer, schmerl}@cs.cmu.edu

***Abstract*—In many scientific fields, simulations and analyses require compositions of computational entities such as web-services, programs, and applications. In such fields, users may want various trade-offs between different qualities. Examples include: (i) performing a quick approximation vs. an accurate, but slower, experiment, (ii) using local slower execution environments vs. remote, but advanced, computing facilities, (iii) using quicker approximation algorithms vs. computationally expensive algorithms with smaller data. However, such trade-offs are difficult to make as many such decisions today are either (a) wired into a fixed configuration and cannot be changed, or (b) require detailed systems knowledge and experimentation to determine what configuration to use. In this paper we propose an approach that uses architectural models coupled with automated design space generation for making fidelity and timeliness trade-offs. We illustrate this approach through an example in the intelligence analysis domain.**

I. Introduction

When software engineers compose existing components into larger systems, they have to make decisions about component selection from component repositories. These decisions are often based on a fixed set of quality trade-offs, where engineers aim for an optimal choice in some trade-off space for some particular context. However, increasingly, compositions may be reused in different contexts where the original trade-off decisions may not make sense, and other choices of components may in fact be more suitable. This problem manifests itself most particularly in domains where compositions are shared with a large community of users, such as in intelligence analysis, medical informatics, or scientific computing.

As an example, consider an intelligence analysis composition that analyzes social-network data for interesting patterns. An analyst who is more concerned about accuracy than time may choose components that use complete information and use complex algorithms to get a nuanced and accurate analysis of the data being examined. Such a complete analysis may take on the order of days or weeks, but can lead to information about the roles, knowledge, locations, and relationships of important actors in that set of data.

Another analyst may want to perform a similar analysis in a situation where information is changing rapidly and timeliness of an answer is important. While the steps to perform the analysis might be the same, the choice of components to perform them in this case will likely differ from those that the first analyst chose. For example, the second analyst may favor quick, less detailed (or low fidelity) analyses over a long and complete (or high fidelity) response. Rather than reuse the original workflow, they are forced to create a new composition specifically for their context, even though the steps are similar. Ideally, the second analyst should be able to reuse the original composition but indicate his trade-off decisions to quickly tailor the composition to his context.

In fact, in domains where composition reuse is common, there may exist multiple (versions of) components with different fidelities or components may be have configuration options to provide different levels of response. Navigating this trade-off space and choosing components to use in a particular context is a complex problem with which software engineers have some difficulty. For domains such as life-sciences and bioinformatics — where composition is usually performed by "professional end-user developers" [23] — making such trade-offs is understandably even harder as the process involves significant analysis and coding skills.

In this paper we explore how architecture models can be used to automate the appropriate configuration of compositions to support trade-offs between different levels of fidelities. The approach takes advantage of the fact that in a given composition, there can be many component realizations having different fidelity and time properties that can be used to realize alternative compositions. We call these compositions *abstract compositions*. Together with component repositories and fidelity/time information we use the Alloy model generator [1] to explore concrete realizations of these abstract compositions that take into consideration a user's fidelity compromises, and then estimate the time it will take to execute the compositon. The user can then explore the compromises on fidelity, see the corresponding time estimates, and make appropriate choices for the given circumstance.

The main contributions of this paper are: (i) a composition approach that enables trade-offs between fidelity choices and execution time, (ii) estimation of time and fidelity using order information, and iii) using model checking to generate compositions that optimize the trade-offs.

The rest of the paper is organized as follows. In Section II, we expand on the problem and introduce the requirements for providing automated fidelity/time trade-off compositions. In Section III, we describe our approach, which involves passing architectural descriptions of abstract workflows to Alloy that generates concrete workflows matching the desired fidelity points. Section IV gives related work, and Section V provides some discussion and future work.

II. Problem

In many scientific fields, simulations and analyses require computations with varying fidelity expectations. For example, scientists may perform a quick approximation using lesser data, or perform computations with various fidelity trade-offs. In many such domains, composing heterogeneous computational entities, usually in the form of workflows or component assemblies, allows scientists to execute their analyses. In these scenarios, often the fidelity selection of datasets, components, and their configurations determines the timeliness of queries (and vice-versa).

However, such component compositions are difficult to specify. There are often inter-component constraints that may lead to mismatches [15]. There may also exist multiple alternative configurations, with a different set of parameters that may provide results with different execution times and fidelities. It is even more difficult when a user has to make trade-offs between fidelity and timeliness choices to select a *component configuration* that meets expectations since the number of possible configurations grows combinatorially. Furthermore, user's fidelity selection may not directly match a configuration and the nearest approximation may be required. Existing composition approaches do not provide a good mechanism to support such fidelity and timeliness trade-offs [8].

As an illustration, consider an example in the field of military intelligence (also called edge analytics) where soldiers rely on analysis and simulations to guide their operations. Today, such analytic capabilities are provided by tools and mechanisms that can transform information sources captured as unstructured input (e.g., incident reports, news sources, miscellaneous geo-spatial data) into complex network models that aid sophisticated analysis such as situational awareness, key entities, fact identification, and what-if exploration [5].

A common querying scenario is when a soldier observes suspicious activity and sends an incident report (see example below) to an operating base, where analysts and other experts can analyze the incident and respond back with a report.

> **Incident Report:** Lt. Col. Liz Abreams (Date: 2/16/2011) Set up sensor alert at checkpoint zulu-1, border crossing between Talodi and Malakal. Position sensor picked up 15 vehicles. Darfur escapees. Overcast. Positive ID on LP 6VES512. Orange. Passengers were Hasim Makul, Hassan Sayid Deng, Jon Deng, and Mary Okulo. Visible knifes. Possible narcotics."
> **Query:** Should we detain? Will maintain position till 1800.

In order to assess the situation and to answer the soldier's question, an analyst in a forward operating base must decide whether the new information leads to any significant changes in the existing network structures in that geographical area. An illustrative computational workflow (shown in Figure 1) for this query involves: (i) processing this incident report along with the network data, (ii) converting it into a graph-based model and (iii) using network algorithms to create and visualize the impact of the new event.

In this domain, a typical network contains millions of nodes with information about people, events, and locations. Therefore, it may save significant time to pre-process and cache this data at the expense of using potentially stale information. Other fidelity variations include: (a) reducing the quantity of data based on dimensions such as time (e.g., only consider this year, vs. consider all years), (b) space (e.g., only consider sources associated with Darfur and Sudan), (c) source (e.g., only consider sources from local reports), and (d) using faster approximations vs. slower but accurate algorithms. These, and other fidelity choices, may lead to different component assemblies with different execution times.

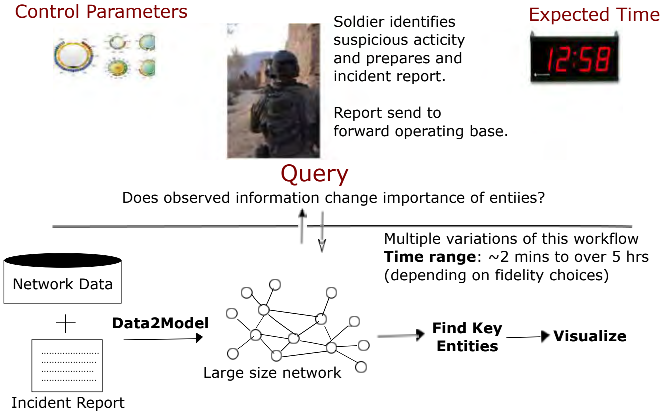

Control Parameters
Soldier identifies suspicious acticity and prepares and incident report.
Report send to forward operating base.
Expected Time
12:58
Query
Does observed information change importance of entiies?
Multiple variations of this workflow
Time range: ~2 mins to over 5 hrs
(depending on fidelity choices)
Network Data
+
Data2Model
Incident Report
Large size network
Find Key Entities
Visualize

Fig. 1: Querying based on fidelity vs. timeliness.

A variation of the workflow from Figure 1 is illustrated in Figure 2 that has fidelity reductions in terms of using cached data with a faster approximation algorithm for computing key-entities. The end user (here, an analyst) provides the control parameters or fidelity expectations that can inform him about the expected execution time and help in the generation of the right computation assembly that serves his operational needs. While the workflow in Figure 1 takes more than 5 hours to execute, the one in Figure 2 takes about 2 minutes. This dramatic time saving is achieved by approximating the results by using a slightly older, cached network and a faster algorithm that uses a subset of the networks that deals with relationships between people from the Sudan network data (instead of using a collection of other relationships such as knowledge, resources, geospatial or temporal information, etc. that can provide a detailed, but slower, analysis).

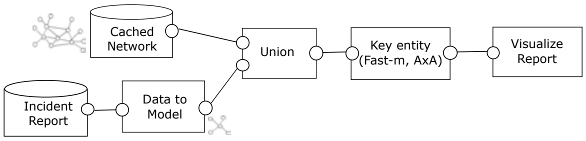


Fig. 2: A variation of the workflow in Figure 1.

The above example highlights interesting challenges in providing flexibility to the information analyst to process data and provide answers to questions in a timely manner. These challenges include:

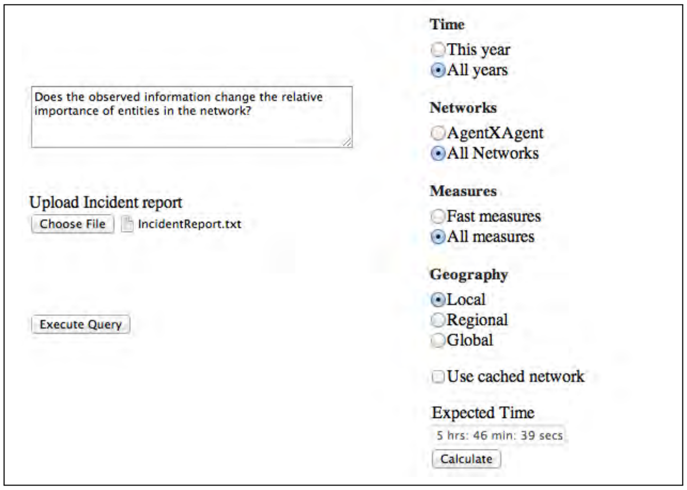
1. Overwhelming component choice: How does the intelligence analyst map the abstract steps that must be followed to produce an answer to the actual program elements and services that are available? As we have reported previously [10], many domains including intelligence analysis have an abundance of components to choose from that provide similar functionality, and that can be parameterized for slightly different results. To map the abstract steps of the workflow to concrete elements, users in most cases resort to familiarity rather than applicability in the face of a lot of options.

2. Multiple fidelity dimensions: Analysts may want quick answers to questions, but they also need to understand what they are giving up for speed. In fact, there are typically multiple dimensions that need to be compromised for speed. In the example above, an analyst can choose to filter the data being used on several dimensions (time, geography), choose to use network analyses that do not follow potential paths in the network (and therefore cannot determine important attributes like grouping or connectedness), or choose to use only certain aspects of the networks (e.g., to focus on social networks and ignore knowledge or belief networks).

3. Inter-component dependencies: Further complicating the choice of components and fidelities is the fact that dependencies exist between the options in the composition. For example, choosing to use only the most recent data prohibits the use of trend analysis later in the workflow. Choosing a component that produces a certain restricted format of data may constrain the choice of downstream components that can be used.

III. Approach

An approach that helps to automate the above choices is required to simplify the composition of components for end users. In this section we describe an overview of our approach for providing a system that allows end users to explore the fidelity and time trade-offs in component compositions.



Does the observed information change the relative importance of entities in the network?

Upload Incident report

Choose File IncidentReport.txt

Execute Query

Time
○ This year
● All years

Networks
○ AgentXAgent
● All Networks

Measures
○ Fast measures
● All measures

Geography
● Local
○ Regional
○ Global

□ Use cached network

Expected Time
5 hrs: 46 min: 39 secs

Calculate

Fig. 3: A simple UI to perform fidelity time trade-offs.

A. Overview

To address the requirements above, we use an approach, illustrated in Figure 4, that allows a user to choose the fidelity options they are willing to make and receive information about the estimated time that the composition will take to perform a query. An expert analyst (who is a domain-expert) develops an abstract component composition describing the steps that are required to process a query. A component repository contains the instantiations of all components that provide concrete realizations of the steps in the composition, along with their fidelities and timing profile. To perform a query, a user fills in the fidelity options they are willing to compromise on. (See Figure 3).

We model the user's choices, the abstract compositions, and the concrete components in the Alloy modeling language [19].

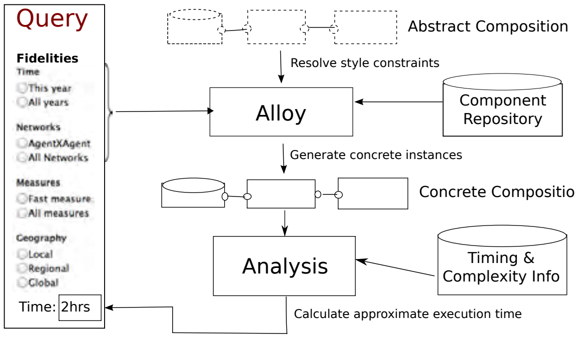


Fig. 4: Generating concrete workflow and timing.

An Alloy model is a collection of signatures and constraints that describes a set of structures, for example: all the possible configurations of a web application, or all the possible assemblies of a set of components that follow composition rules. The Alloy Analyzer is a solver that takes the constraints of the model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures and to check properties of the model by generating counterexamples. Specifically, for our problem, we use Alloy's model-generation capability to generate compositions required to answer queries from analysts (described in more detail below).

In our approach, we consider two levels of compositions: (i) abstract compositions that are defined by an assembly of components that have some high-level properties, constraints and functions, and (ii) concrete compositions that are an assembly of computational elements that implement those functions. The high-level choices made by the end-user help in the selection of the abstract components. The Alloy model generator generates concrete compositions that satisfy composition and fidelity constraints. This concrete composition is then further analyzed (as explained later), using complexity and timing information, to generate an approximate execution time that can be given back to the analyst, who can then either choose different fidelity options, or run the concrete composition.

When the analyst chooses to execute a concrete composition, it is translated into a script that can be executed. In our current prototype, we generate a BPEL script that uses components derived from existing intelligence analysis tools deployed on the SORASCS platform [9].

B. Architecture representation of compositions

In our earlier work, we proposed compositions as end-user architectures [10] that can be explicitly represented as architectural models defined in a domain-specific architectural style. Such architecture models can be used not only for various analyses, but they can also generate executables. Such architectures form the basis of composition and fidelity analysis. For instance, here we represent end-user compositions in SCORE [16] — a dataflow based style that is customized for the intelligence analysis domain.

We adopt the architecture analysis approach from Kim et al. [20] where architectural types are specified in Alloy as signatures and constraints (based on fidelity and other architectural properties). These are analyzed for type checking and model generation. Compositions in SCORE are converted to the Alloy specification language that is based on first-order and relational calculus well suited for representing abstract compositions.

While details about modeling architectures and their analysis in Alloy can be found in [20], we walk you through an example modeling scenario in Alloy. Listing 1 shows a snippet of the architecture model in Alloy (defined as a configuration of components and connectors). We extend this specification to define two types of components — abstract and concrete. A snippet of mapping between abstract and concrete components is shown in Listing 2.

Listing 1 : Architecture specification in Alloy.

```
...
abstract sig Architecture {
    comps: set Component,
    conns: comps lone -> lone comps /*connectors modeled as
        a relation between components */
}{
    no (iden & ^conns) /* no cycles */
    all c: comps | comps in c.*conns + c.*~conns /* fully
        connected*/
}
/* Data dependencies are satisfied */
pred WellFormedArch (arch: Architecture) {
    all c1, c2: arch.comps | (c1 -> c2) in arch.conns => c1.
        output.data = c2.input.data
}
...
```

Listing 2 : A mapping between abstract and concrete architectures.

```
...
sig RealizationMap {
    absArc: one AbstractArchitecture,
    concArc: one ConcreteArchitecture,
    impls: ConcreteComp one -> one AbstractComp
}
{all cc,ca: Component | (cc -> ca) in impls => (cc in
    concArc.comps) and (ca in absArc.comps) /*Mapping only
    existing components*/
    all ca:absArc.comps | some cc:concArc.comps | (cc->ca)
        in impls /*each abstract has a concrete one*/
    all cc:concArc.comps | some ca:absArc.comps | (cc->ca)
        in impls /*each concrete has an abstract one*/
}
...
```

Concrete models are constrained to satisfy both the structure of the abstract model and the fidelity properties selected by the end user, represented in Alloy as properties and predicates to be satisfied in the model generation phase. A simple example of fidelity properties is described in Listing 3.

Listing 3 : Specifying fidelities.

```
...
/*Network types*/
one sig AxA, AllNetworks extends NetworkTypeFidelity{}
/*Data Size*/
one sig ThisYear, AllYear extends DataSizeFidelity{}
/*Speeds*/
one sig FastOnly, AllSpeeds extends SpeedFidelity{}
/*Data Regions*/
one sig Local, Regional, Global extends DataRegionFidelity{}
...
/*Pred to check if types are satisfied*/
pred satisfiesDimensionType() {
    all f: Fidelity | (f in Speed => satisfiesSpeedType[f])
        and (f in NetworkType => satisfiesNetworkType[f])
        and (f in DataSize => satisfiesDataSize[f]) and (f
        in DataRegion => satisfiesDataRegion[f])
}
```

C. Generate concrete compositions

As we discussed before, we use Alloy's model generator to create concrete models representing compositions. The concrete assembly is constrained to follow data-mismatch [15] and fidelity constraints. As an illustration, see Figure 5 where a high level workflow is mapped to a concrete composition and modeled using types (or signatures) comprising of components, interfaces, their properties and the constraints. We do this by representing the abstract and concrete component vocabulary, along with a mapping function in Alloy and using its model generation capability that generates concrete instances given a set of high-level components and their properties.

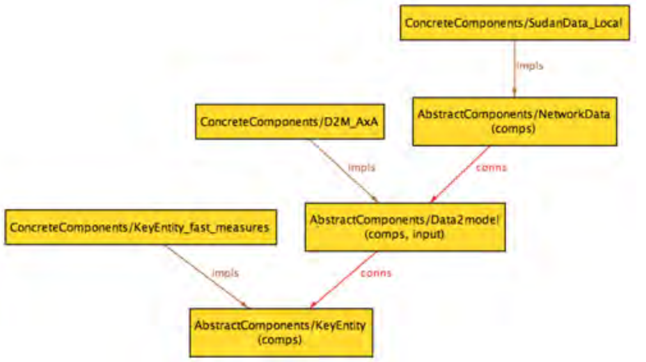

ConcreteComponents/SudanData_Local
impls
ConcreteComponents/D2M_AxA
AbstractComponents/NetworkData (comps)
impls
conns
ConcreteComponents/KeyEntity_fast_measures
AbstractComponents/Data2model (comps, input)
impls
conns
AbstractComponents/KeyEntity (comps)

Fig. 5: An illustration of a mapping between abstract and concrete workflows in Alloy.

The abstract workflow in Figure 5 comprises three functions with varying input and output data requirements and fidelity constraints. This is mapped into a concrete workflow that includes services for individual functions and additional components for data translation and data fetching to generate a sound composition. The Alloy snippet in Listing 4 shows the function that is run by Alloy to create instances of concrete models given an abstract model within a specified scope of number of objects per signature (e.g., by using the "run for 15" command). A simplified concrete model is shown in Figure 5 (with additional details such as ports, properties and mappings turned off).

Listing 4 : Generating model instances in Alloy.

```
...
pred showConcreteFromAbstract{
    one RealizationMap
    one _AbstractArch
    _AbstractArch in RealizationMap.absArc
    one ConcArchitecture
    ConcArchitecture in RealizationMap.concArc
    Component in (ConcArchitecture.comps + AbsArchitecture.
        comps)
     SatisfiesFidelities[ConcArchitecture, fast, AxA,
        ThisYear, Local]
}
...
run showConcreteFromAbstract for 15 but 1
    AbstractArchitecture,
1 ConcreteArchitecture
```

D. Performance Analysis

Next, we perform an analysis that computes the execution time for the concrete compositions. We use properties of the networks and the complexity of the algorithms to approximate the execution times for components in the workflow, and aggregate them into the overall execution time. These approximations of execution time help end users to make an

TABLE I: A selection of centrality metrics

Metrics	Description	Reference	Complexity	Class
Degree Centrality	Number of neighbors	[25]	O(m)	fast
Hubs and Authorities	Power centers and connectors	[21]	O(m)	fast
Eigenvector Centrality	Power centers	[3]	$\sim$O(n^2)	medium
Clique Count	Part of dense groups	[4]	$\sim$O(nm)	medium
Cognitive Demand	Involvement in many activities	[6]	O(nm)	medium
Situation Awareness	Overview of activities	[18]	O(nm)	medium
Betweenness Centrality	Control communication flow	[17]	O(nm+ n^2log n)	slow

informed choice about the timeliness of compositions based on selected fidelities.

To illustrate the approach, consider "Generate Key Entities." *Key entities* refer to important agents in a network. Social network analysis has developed a wide array of metrics to describe the position of individuals within a graph-like structure consisting of nodes and edges [25]. As different metrics focus on different aspects of network importance (e.g., centrality), analysts often calculate a set of centrality metrics for identifying and ranking the important nodes in a network. Calculating these metrics and comparing the results with previous results can help the analyst to assess whether the newly observed situation creates significant change in the network.

Table I shows a selection of these metrics with their computational complexity. This enumeration is a selection to illustrate our approach; an actual application scenario consists of many more metrics.

The right column of Table I shows the complexity class that we use to filter the algorithms — n refers to the number of nodes and m to the number of edges. For estimating the calculation time, we first summarize the metrics complexities grouped by complexity class:

$$t_{fast} = O(m)\text{..................................}(1)$$
$$t_{medium} = O(nm + n^2)\text{..........................}(2)$$
$$t_{slow} = O(nm + n^2 logn)\text{...................}(3)$$

Based on a user's explicit or implicit (by timeliness/fidelity options) selection of metrics we can determine the aggregated calculation complexity of all metrics that are included in the calculation. For instance, in order to calculate all metrics, the accumulated calculation complexity is:

$$t_{all} = t_{fast} + t_{medium} + t_{slow}\text{................}(4)$$
$$= O(m + nm + n^2 + n^2 logn)\text{..........}(5)$$

As we know n and m (the number of nodes and edges in the network), we can estimate the actual calculation time in seconds of any combinatorial set of metrics after an initial "calibration" procedure for which we calculate a selected (small) set of metrics on a given network with a given machine, and measure the time in seconds needed for the calculations. It is to be noted that we ignored effects such as paging that may be evaluated for further optimization.

Using a small-world network [26] with 10,000 nodes and 50,000 edges we calculate betweenness centrality. This takes 80 seconds on a regular laptop computer. With this number we can estimate the timeliness of all combinations of metrics.

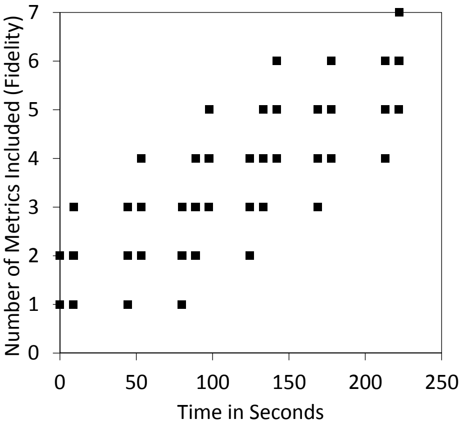


Fig. 6: Timeliness/Fidelity option space.

Even though our example consists of just seven metrics, there are 127 combinatorial options (2^7-1). The complete option space is visualized in Figure 6; some dots represent more than one metrics combination. In case we have no constraints or preferences on metrics this figure tells us that, for example, we can execute up to 5 metrics within 100 seconds.

E. Orchestration Generation

The concrete workflows generated by Alloy provide valid component compositions that obey configuration and respect fidelity preferences. They define a sequence of web-services provided by the SORASCS platform [9]. These concrete compositions are compiled into orchestration scripts and executed using a standard SOA infrastructure. The final result is the output of executing the service-composition. For the composition scenario discussed in the paper, it is a key-entity report that executes a set of centrality metrics (based on user selection) for the analysis query.

F. Implementation

To demonstrate our approach we implemented a prototype web-application (using the example scenario described in Section II) where users can specify their fidelity choices based on expected execution times. These fidelity choices are incorporated in an Alloy model that helps us to generate executable compositions. Our goal was to demonstrate (a) that the approach is feasible, and (b) it can scale to implement fidelity-timeliness trade-offs in realistic time.

For our prototype, we used data about terrorist activities in Sudan, consisting of about 10 years of news reports and incidents. Our data repository consists of about 300Mb-400Mb of plain text reports for each year, which when processed, consists of networks with 379,638 nodes and 15,373,115 edges (with information about people, events, and locations). The fidelity choices (such as caching, reduction in scope, etc.) therefore have a major impact on the processing time of the compositions.

We ran our prototype on a 3 GHz Intel Core 5 machine with 4GB RAM. Even for a simple composition scenario, our approach allows fidelity variations that lead to execution times ranging between 2 mins 23 secs to more than 5 hours. The total execution time for Alloy model-generation varies between 1600 milliseconds and 2400 milliseconds, which is almost

TABLE II: Fidelity vs. Timeliness results

Fidelity Choices			Execution Time
Data	**Meta-networks**	**Algorithm**	
Current year	Agent-Agent only	Fast metrics only	0 hrs: 2 min: 23 secs
Current year	Agent-Agent only	All metrics	0 hrs: 3 min: 54 secs
Current year	All relationships	All metrics	0 hrs: 4 min: 14 secs
All year	Agent-Agent only	Fast metrics only	2 hrs: 26 min: 49 secs
All year	All relationships	All metrics	5 hrs: 46 min: 39 secs

negligible given the much larger execution time for the entire composition. Table II shows the common variation points and their impact on execution times.

IV. Related Work

There has been some significant work towards analysis of compositions. Examples include: QoS (Quality of Service) based analysis for service compositions [12], analysis of time constraints based on Petri Nets [24] and soundness checks [22] for BPEL orchestrations. Furthermore, tools such eflow [11] by Fabio Casati have addressed issues with dynamic service compositions. However, most of these analytic approaches are geared towards static design choices where there is limited support for trade-offs in the fidelity and timeliness space.

A related domain where such trade-offs have been relevant is product-line engineering. There has been some work towards automatic prediction of execution time based on feature composition by Siegmund et al. [14]. Another related work in the product-line compositions has been the Clafer tool [13] that extends Alloy-based simulations with multi-objective calculations to determine the best set of features in a product line. In our work we have addressed trade-offs between (various dimensions of) fidelity and timeliness and have automated it to be able to synthesize executable code. Similar to Bagheri and Sullivan [2], we use architecture as a basis for such reasoning and code-generation.

V. Conclusion and Future work

We demonstrated an architecture-based approach to make fidelity vs. timeliness trade-offs in a realistic domain. As a first step, we presented a simple scenario where we generate compositions based on performance profiles of components. In this case study we used a small number of metrics resulting significant, but relatively small option space. However, we are confident that more complex trade-offs can be handled by this model generation technique. As a future step, we plan to expand this to other domains and other dimensions of fidelity trade-offs.

A more complicated problem, and a possible future work, is to do the reverse i.e., calculating the fidelity options given the execution time requirements, which is similar to solving multi-objective problems [7]. Also, for this prototype we have not considered all possible model instances generated by Alloy, which may be evaluated for more optimal solutions.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Further support for this work came from Army Research Office under Award No. W911NF1310155. The authors would like to thank Ed Morris, Soumya Simanta, Kathleen Carley, Troy Mattern and Jeff Boelng for their valuable contributions. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. This material has been approved for public release and unlimited distribution. DM-0000706

References

[1] Alloy analyzer. http://alloy.mit.edu/alloy4/.

[2] H. Bagheri and K. J. Sullivan. Pol: specification-driven synthesis of architectural code frameworks for platform-based applications. In *GPCE*, pages 93–102, 2012.

[3] P. Bonacich. Factoring and weighting approaches to status scores and clique identification. *Jour. of Math. Sociology*, pages 113–120, 1972.

[4] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[5] K. M. Carley and J. Pfeffer. *Dynamic Network Analysis (DNA) and ORA*. Advances in Design for Cross-Cultural Activities (Part 2), D. D. Schmorrow, D.M. Nicholson (eds), CRC Press, 2012.

[6] K. M. Carley, J. Pfeffer, J. Reminga, J. Storrick, and D. Columbus. Ora users guide 2013, 2013.

[7] K. Deb. Multi-objective optimization. *Multi-objective optimization using evolutionary algorithms*, pages 13–46, 2001.

[8] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528 – 540, 2009.

[9] B.R. Schmerl et al. SORASCS: a case study in SOA-based platform design for socio-cultural analysis. In *ICSE*, pages 643–652. ACM, 2011.

[10] D. Garlan et al. Foundations and tools for end-user architecting. In *Monterey Workshop 2012*, LNCS, pages 157–182. Springer, 2012.

[11] F. Casati et al. Adaptive and dynamic service composition in *eflow*. In *Seminal Contr. to Inf. Systems Eng.*, pages 215–233. 2013.

[12] J. Cardoso et al. Quality of service for workflows and web service processes. *J. Web Sem.*, 1(3):281–308, 2004.

[13] M. Antkiewicz et al. Clafer tools for product line engineering. In *SPLC Workshops*, pages 130–135, 2013.

[14] N. Siegmund et al. Predicting performance via automated feature-interaction detection. In *ICSE*, pages 167–177, 2012.

[15] P.V. Elizondo et al. Resolving data mismatches in end-user compositions. In *IS-EUD*, pages 120–136, 2013.

[16] V. Dwivedi et al. An architectural approach to end user orchestrations. In *ECSA*, pages 370–378. Springer-Verlag, 2011.

[17] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40:35–41, 1977.

[18] J.M. Graham, M. Schneider, and C. Gonzalez. Report social network analysis of unit of action battle laboratory simulations (cmu-sds-ddml-04-01). carnegie mellon university, social and decision sciences., 2004.

[19] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.

[20] J.S. Kim and D. Garlan. Analyzing architectural styles. *Journal of Systems and Software*, 83:1216–235, July 2010.

[21] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[22] F. Puhlmann and M. Weske. Interaction soundness for service orchestrations. In *ICSOC*, pages 302–313, 2006.

[23] J. Segal. Some problems of professional end user developers. In *VL/HCC*, pages 111–118, 2007.

[24] W. M. P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In *BPM*, pages 161–183, 2000.

[25] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[26] D. J. Watts and S. H. Strogatz. Collective dynamics of'small-world'networks. *Nature*, 393(6684):409–10, 1998.

Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming

Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich

Institute for Software Research
School of Computer Science
Carnegie Mellon University
{sunshine, jdh, aldrich}@cs.cmu.edu

Abstract. Application Programming Interfaces (APIs) often define object protocols. Objects with protocols have a finite number of states and in each state a different set of method calls is valid. Many researchers have developed protocol verification tools because protocols are notoriously difficult to follow correctly. However, recent research suggests that a major challenge for API protocol programmers is effectively searching the state space. Verification is an ineffective guide for this kind of search. In this paper we instead propose Plaiddoc, which is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. We compare Plaiddoc to a Javadoc control in a between-subjects laboratory experiment. We find that Plaiddoc participants complete state search tasks in significantly less time and with significantly fewer errors than Javadoc participants.

1 Introduction

Many Application Programming Interfaces (APIs) define object protocols, which restrict the order of client calls to API methods. Objects with protocols have a finite number of states and in each state a different set of method calls is valid. Protocols also specify transitions between states that occur as part of some method calls. A client of such a library must be aware of the protocol in order to use it correctly. For example, a file may be in the open or closed state. In the open state, one may read or write to a file, or one may close it, which causes a state transition to the closed state. In the closed state, the only permitted operation is to (re-)open the file.

Files provide a simple example of states, but there are many more examples. Streams may be open or closed, iterators may have elements available or not, collections may be empty or not, and even lowly exceptions can have their cause set, or not. More than 8% of Java Standard Library classes and interfaces define protocols, which is more than three times as many as define type parameters [1].

Protocols are implemented in mainstream languages like Java with low-level constructs: the state of an object is tracked with boolean, integer, or enum fields;

violations are checked explicitly and cause runtime exceptions like IllegalStateException; and constraints are specified in prose documentation. It is perhaps unsurprising, therefore, that APIs with protocols are difficult to use. In a study of problems developers experienced when using a portion of the ASP.NET framework, three quarters of the issues identified involved temporal constraints [19]. Three recent security papers have identified serious vulnerabilities in widely used security applications resulting from API protocol violations [14, 4, 28].

Many researchers have developed protocol checkers which are designed to make it easier for programmers to correctly use APIs with protocols (e.g. [3, 10, 13]). These tools require programmers to specify protocols using alias and typestate annotations that are separate from code. To automate the annotation process, several tools mine protocol specifications using dynamic analysis [8] or static analysis [2, 36]. A recent survey of automated API property inference techniques described 35 inference techniques for ordering specifications [24].

However, the qualitative studies described in [31, ch.3] found that programmers using API protocols spend their time primarily on four types of searches of the protocol state space. Protocol checker output is unlikely to help programmers perform many of these searches.

Instead, in this paper we introduce a novel documentation generator called Plaiddoc, which is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. Plaiddoc is extracted automatically from the standard Javadoc annotations plus new Plaiddoc specifications. Plaiddoc is named for the Plaid programming language [32], which embeds similar state-oriented features, and from which Plaiddoc could, in principle, be automatically generated. We evaluate Plaiddoc against a Javadoc control in a 20-participant between-subjects laboratory experiment.

The experiment attempts to answer the following five research questions:

RQ1 Can programmers answer state search questions more efficiently using Plaiddoc than Javadoc?

RQ2 Are programmers as effective answering non-state questions using Plaiddoc as they are with Javadoc?

RQ3 Will programmers who use Plaiddoc answer state search questions more correctly than programmers who use Javadoc?

RQ4 Will programmers get better at answering state search questions as they get more practice?

RQ5 Are programmers who use Plaiddoc better than programmers who use Javadoc at mapping general state concepts to API details?

All of the tasks performed by participants asked participants to answer a question. We therefore use the words task and question interchangeably in the rest of this paper. Most of these questions were instances of four state search categories discovered in two earlier, qualitative studies [31]. Some of the questions were not state related and were chosen to benefit Javadoc. Task ordering was alternated to measure learning effects, and a post-study quiz was administered to gauge concept understanding.

Participants using Plaiddoc completed state tasks in 46% of the time it took Javadoc participants, but were approximately equally fast on non-state tasks. Plaiddoc participants were also 7.6x less likely to answer questions incorrectly than Javadoc participants. Finally, Plaiddoc and Javadoc participants were approximately equally able to map state concepts to API details. Nevertheless, our overall results suggest that Plaiddoc can provide a lightweight mechanism for improving programmer performance on state-related tasks without negatively impacting traditional tasks.

More broadly, the results of this study also provide indirect support for several programming language design choices. This study provides quantitative evidence for the productivity benefits of type annotations as documentation and state-oriented language features.

2 Background and Related Work

The seminal paper entitled "Why a diagram is (sometimes) worth ten thousand words," [21] introduces a computational model of human cognition to compare informationally equivalent diagrams and text. They demonstrate in this model that solving math and physics problems with text-based information can require many more steps than solving the same problems with diagrams. The most important difference between the diagram steps and text steps is that much more effort in text is spent searching for needed details. One particularly noteworthy reason for the search difference is that diagrams often collocate details that are needed together.

Larkin and Simon's theory has been effectively applied to many other (non-diagramatic) information contexts. For example, Chandler shows in a series of experiments that integrated instructional material and the removal of non-essential material can facilitate learning in a variety of educational settings [5] . There are many more closely related examples: Green [15] develops cognitive dimensions to evaluate visual programming languages, the GOMS [20] model has proven effective at predicting user response to graphical user interfaces (GUIs), and MCRpd [34] models physical representations of digital objects.

The results of two studies of API design choices are best understood through Larkin and Simon's search lens. It is easier for programmers to use constructors to create instances than factory methods, because constructors are the default and are therefore the start of any search [11]. Methods that are located in the class a programmer starts with are easier to find than methods in related classes [30]. The impact of small design changes shown in these papers emphasizes the importance of information seeking on API usability, and suggests that a similar impact may be possible with other small interventions.

All of this research suggests that there is an opportunity to modify an API artifact to create an informationally equivalent alternative that will improve programmer performance with protocol search. Which artifact? Which changes will be most effective? To answer these questions it is useful to look at the interventions that have proven effective with other complex APIs.

One effective way to learn to use an API is to find a related example. A study of programmers using reusable Smalltalk GUI components and found that participants “relied heavily on code in example applications that provided an implicit specification for reuse of the target class.” The significance of examples encouraged researchers to develop example repositories to enable programmers to find examples easily [23, 37]. Unfortunately, the effectiveness of these repositories was limited by the retrieval mechanism which required too much (and too complex) input from programmers.

More recently, MAPO [38] and Strathcona [18] automatically retrieve examples from the structure of the program the programmer is writing. In a controlled experiment, participants using MAPO produced code with fewer bugs than participants in other conditions. This result is notable because it shows that API interventions can produce higher quality responses, not just more rapid responses.

The eMoose IDE plugin has proven similarly useful to developers using complex API specifications [9].The eMoose tool pushes directives—rules required to use a method correctly—to the method invocation site. The concrete rules that make up a protocol (e.g. one cannot call setDoInput on a connected URLConnection) are examples of directives. Dekel’s evaluation of eMoose demonstrated significant programmer performance improvements during library-usage tasks (including one library with a protocol).

Unfortunately, examples and directives are labor intensive for API designers to produce. In large complex APIs it is often impossible to generate examples for every possible use case. Even after they are produced, it is hard to keep them in sync with the API as it changes, because there is no mechanism to enforce conformance. Examples can also serve as a crutch toward learning, and the most effective students learn to generate their own examples [6].

The design of Plaiddoc is inspired by all of the research discussed in this section. We modify Javadoc to produce an informationally equivalent documentation format aimed at facilitating speedier state search. Plaiddoc is generated from specifications whose conformance with code can be checked automatically. Plaiddoc specifications, like eMoose directives, are co-located with each method. The specifications themselves contain just the right state details so programmers can generate their own examples of correct API usage. The details of the Plaiddoc design are discussed in the next section.

3 Plaiddoc

To follow the rest of this paper, it is important to understand the design of Plaiddoc. To do so, it is necessary to first explain Javadoc. Javadoc is a tool for generating HTML documentation for Java programs. The documentation is generated from Java source code annotated with “doc comments” which contain both prose description and descriptive tags which tie the prose to specific program features. For example, a doc comment on a method will describe the method in general and then provide tags and associated comments for the parameters, the return value, and/or any exception the method throws.

The webpage generated by Javadoc for a class has six parts. The top and bottom contain navigation elements which allow the reader to quickly browse to related documentation. The class description appears below the navigation elements at the top of the page. It states the name of the class and links to superclasses and known subclasses. It then follows with an often long description which can include: the purpose of the class, how it is used, examples of use, class-level invariants, relationships to other classes, etc.

After the class description, the page includes four related elements: the field summary, method summary, field details, and method details. The field summary is a table containing the modifier, type, name, and short description of each public field sorted in alphabetical order. The method summary is extremely similar: it shows the modifier, return type, method name, type and name of all parameters, and short method description in alphabetical order. The field and method details show each field (or method) in the order they appear in the source file with the full description including historical information and any tags.

The Plaiddoc generated webpage maintains all of the look and feel of the Javadoc page. The fonts, colors, and visual layout are identical. However, the method summary section is restructured and extra information is added to the method details section. The full ResultSet page is available on the web.[1] The screenshot shows the method summary for the top-level Result state and the Open state.

As in Plaid, methods in the summary are organized by abstract state. In Javadoc, there is one table containing all of the methods of a class, while in Plaiddoc there is one table per abstract state. For example, the Disconnected state of URLConnection has a table containing all of the methods available in it, including setDoInput and connect.

One important rule we followed when designing Plaiddoc is that there is exactly one Plaiddoc page per Javadoc page. This rule ensures that the any observed differences between participants using Plaiddoc and Javadoc is a consequence of Plaiddoc's extra features and not the result of differences in page switching. There are two consequences of this rule: 1) All of the possible states of single Java class appear in the same Plaiddoc page.[2] 2) Multi-object protocols appear in multiple Plaiddoc pages. Six of the tasks in this study involve the Timer and TimerTask classes which impose a multi-object protocol. In these tasks, Javadoc participants were given two pages and Plaiddoc participants were given two pages.

An automatically generated diagram which shows all of the states of the class and where the particular state fits in, appears above each state table. The current state is bolded and italicized, while other states are displayed in the standard font. This diagram is *primitive*; it does not contain extensive capabilities like hyperlinks from state names to state tables, collapsing/expanding children, transition arrows, or even a nice graphical look. The diagram is primitive for

[1] `http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/PlaiddocResultSet.html`

[2] e.g. The "Open" and "Closed" states of ResultSet appear on a single page.

three reasons: 1) Plaiddoc was designed for this experiment, and was therefore not polished for use outside the laboratory. 2) More capabilities gives participants more potential paths to solve tasks and thus introduces variation into the study. 3) If one adds features it is harder to understand which particular features are important or unimportant. Plaiddoc was designed with the minimum set of features we believed would be an effective group.

The Plaiddoc page also contains two new columns in the method details table. These columns are state preconditions and postconditions. The only valid predicates are state names, state names with a parameter, or combination of the two separated by the AND or OR logical operators. For example, "Disconnected," "Scheduled task," and "Updatable AND Scrollable" are valid preconditions or postconditions but "value $>$ 0" is not. The same information is added to the method summary. The state to which a method belongs is an implicit precondition for that method. For example, the close method lists no preconditions, but since it belongs to the Open state, the ResultSet must be in the Open state to call the close method.

To generate a Plaiddoc class page, the Plaiddoc tool requires three inputs: the class's Javadoc page, a JSON file specifying the state relationships of the class, and a JSON file containing preconditions and postconditions for each method and mapping methods to states. Sample JSON files are available on the web.[3]

The JSON files are very simple. The state file must contain a single object whose fields are states, each of which must contain either an "or-children" or "and-children" field. These "children" fields are arrays containing state names, which in turn must be defined in the same file. The methods file must contain an array of method objects which contain four fields: "name" (including parameter types to distinguish statically overloaded methods), "state" (which must map to a state defined in the state file), "pre" for preconditions, and "post" for postconditions.

It is important to map the features of Plaiddoc just described to concepts, in order to understand the implications of the experiment described here on other research (e.g. the Plaid language itself). Plaiddoc organizes methods by state instead of by class, by separating the method summary table by state. Plaiddoc makes state transitions explicit when state postconditions differ from preconditions. The Plaiddoc preconditions and postconditions make use of state-based type specifications. Finally, rich state relationships are displayed to programmers at the top of each method table. See e.g. the "State relationships" box.

4 State search categories

As we mentioned in Section 1, an earlier two-part qualitative study of the barriers programmers face when using APIs with protocols feeds directly into the methodology of the study in this paper [31, ch. 3]. In the first part of that study, we mined the popular developer forum StackOverflow for problems developers

[3] `http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/Car_States.json` and `http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/Car_Methods.json`

have using APIs with protocols. In the second part, they performed a think-aloud observational study of professional programmers in which the programers worked through exactly the problems uncovered in the first part.

In this second part, they analyzed each task, by assigning task time to participant questions or comments and performing open coding on the transcript. This analysis showed that programmers in spent 71% of their total time answering instances of four question categories. We list here each general category followed by two specific instances of that category drawn from the study transcripts:

A What abstract state is an object in?
– "Is the TimerTask scheduled?"
– "Is [the ResultSet] x scannable?"
B What are the capabilities of an object in state X?
– "Can I schedule a scheduled TimerTask?"
– "What can I do on the insert row?"
C In what state(s) can I do operation Z?
– "When can I call doInput?"
– "Which ResultSets can I update?"
D How do I transition from state X to state Y?
– "How do I get off the insert row to the current row?"
– "Which method schedules the TimerTask?'

These search problems are all specific to protocols, and therefore the protocol tasks are dominated by state search. Most of the tasks performed by participants in this study are instances of these general categories.

5 Methodology

The experimental evaluation of Plaiddoc uses a standard two by two between-subjects design, with five participants in each of the four conditions. The experiment compares Plaiddoc to a Javadoc control and presents two task orderings to measure learning effects. The recruitment, training, experimental design, tasks, and post-experiment interview are presented in the following sections. All of the study materials can be found in Appendix C [31].

5.1 Recruitment

All 20 participants were recruited on the Carnegie Mellon campus. Half of the participants responded to posters displayed in the engineering and computer science buildings. The other half were solicited in-person in a hallway outside classrooms which typically contain technical classes. Participants were screened for Java or C# knowledge and experience with standard API documentation. Participants were paid $10 for 30-60 minutes of their time. The 20 participants that made it past the screening all completed the study.

Twelve of the participants were undergraduate students, all of whom were majoring in computer science, electrical and computer engineering, or information

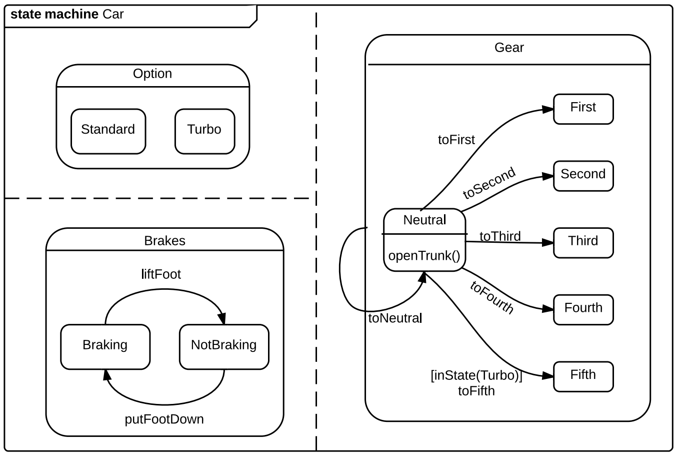


Fig. 1. Car state machine used for participant training.

systems. The other eight were masters students in information systems or computer engineering programs. Eleven students had no professional programming experience outside summer internships, five students had one year of full-time professional experience, and four had more than one year of experience.

5.2 Training

After signing consent forms, participants were given approximately 10 minutes of training. Every participant, regardless of experimental condition, received exactly the same training. The training was read from a script to help ensure uniformity.

All participants were familiar with Javadoc, but the training included an explanation of both Javadoc and Plaiddoc to ensure baseline knowledge in both formats. The goal of this study is to compare the impact of the documentation formats on state search tasks, not the impact of training. Therefore, we kept training consistent to avoid a confounding factor. All of the state concepts are first taught via UML state machines, then Javadoc, then Plaiddoc.

The training materials introduce participants to the basic concepts of object protocols and to the documentation formats used in the study. The training makes concepts concrete using a Car API we constructed for the purpose. Regarding protocols, participants learn:

- that methods are available in some states and not others
- that some methods transition objects between states
- that states can be hierarchical
- that child states can be either or-children or and-children

These concepts were reinforced by asking participants simple, scripted questions about the Car API. The questions were designed to be answerable very quickly by participants. We created a UML state machine (shown in Figure 1), Javadoc documentation, and Plaiddoc documentation for the Car API and these were printed and handed to participants.

The top-level state for Car objects (named "Car") has three and-children, each of which has two or more or-children: *gear* to represent the car's manual transmission, *brakes* to represent whether the car is braking or not, and *option* to represent whether the car has the "turbo" option or not. We used these states to introduce state hierarchy, or-states, and and-states. We introduced transitions via brakes. One can transition to the "Braking" state from the "NotBraking" state by calling the "putFootDown" method. The `openTrunk` method, which does not change the gear state, introduces state-dependent methods. In the example, like in many real-world cars, one can only open the trunk when the car is in the neutral gear.

Like all and-children, the car's three substates are independent, in the sense that changing the gear state has no effect on the braking or option states. However, one unique wrinkle in the example is that the turbo state enables a fifth gear substate of gear that is not available otherwise. The `toFifth` method has two preconditions — the car must be in the neutral gear and it must have the turbo option. In the study tasks discussed later, some of the ResultSet methods also have multiple preconditions.

5.3 Experimental Setup

Participants were asked 21 questions about three Java APIs: 1) Six questions about `java.util.Timer` and `java.util.TimerTask`. We refer to these questions as the Timer questions throughout the rest of this paper. 2) Ten questions about `java.sql.ResultSet`. 3) Five questions about `java.net.URLConnection`. The experimenter read each question aloud and handed the participant a piece of paper with the same question written on it.

Participants were seated in front of a computer, and asked to answer the question by looking at documentation on the computer screen. The experimenter opened the documentation for the participant in a browser window. Both the Javadoc and Plaiddoc documentation were opened from the local file system to present a consistent URL and to prevent network-related problems. The computer screen and audio (speech) were recorded with Camtasia.

Half of the participants were shown standard Javadoc documentation for all questions and half Plaiddoc documentation. Participants were allowed to make use of the browser's text search (i.e. Control-F). However, they were not allowed to use internet resources (e.g. Google, StackOverflow).

We chose a between-subjects design to control for cross-task contamination. Many software engineering studies use within-subjects designs to reduce the noise from individual variability. We guessed based on pilot data that individual variability in our study would be relatively low and we therefore opted for the cleaner between-subjects design. As we will see in §6, the study was sufficiently sensitive to distinguish between conditions so our guess turned out to be accurate.

Questions were asked in batches — all of the questions related to a particular API were asked without interruption from questions about another API. Within each batch, each question was asked in the same order to every participant. However, half of the participants were asked the Timer batch first and half were asked the UrlConnection batch first. The ResultSet batch always appeared second and the remaining batch appeared third. We wanted the Timer and URLConnection batches to each appear last so we could measure the learning effects on those batches. All other ordering was uniform across conditions to avoid unnecessary confounding factors.

The study had a total of four between-subjects conditions: Plaiddoc with Timer first (condition #1), Plaiddoc with URLConnection first (condition #2), Javadoc with Timer first (condition #3), and Javadoc with URLConnection first (condition #4). Participants were assigned to conditions based on the order they appeared in the study. The nth participant was assigned to condition #n modulo 4. Using commonly accepted practice, participants were assigned to conditions pseudorandomly, in the order they arrived. Therefore, there were exactly five participants in each condition.

5.4 Tasks

The 21 questions asked of the participants are shown in Table 1. Sixteen of the questions were instances of the four categories of state search enumerated in §**??**. Since these questions are state specific, we refer to them as the state questions. The remaining five questions were non-state questions, which were designed to be just as easy or easier with Javadoc than Plaiddoc. These questions were not about states or protocols, and we therefore refer to them as the non-state questions.

We selected the state questions with a three-phase process. First, we generated all of the instances of the general categories we could think of for each API. Second, since we did not want the answer or the process of answering one question to affect others, we removed questions which were not independent. Some additional non-independent questions were removed during piloting. Third, we pruned the ResultSet questions to include two instances of each question category by random selection. The study was too long with the full set of ResultSet questions.

The final question set includes three instances of A) "What abstract state is an object in?", five instances of B) "What are the capabilities of an object in state X?", four instances of C)"In what state(s) can I do operation Z?',' and four instances of D) "How do I transition from state X to state Y?" Participants in all conditions were given a glossary listing all of the states of the API in question with a short description of each. Participants were instructed to answer questions in categories A and C with the name of a state from the glossary. In other words, these questions were multiple choice.

The names of states in the glossary matched those in Plaiddoc. The names themselves were taken from the Javadoc as much as possible. We did not want

Table 1. Category, identifier and question text for all of the questions asked of participants in the main part of the study. Questions with identifiers beginning with T involved `java.util.Timer` and `java.util.TimerTask`, R involved `java.sql.ResultSet`, and U involved `java.net.URLConnection`.

Cat.	ID	Question text
T	T-1	How do I transition a Timer Task from the Virgin state to the Scheduled state?
N	T-2	What is the effect of calling the purge method on the behavior of the Timer?
C	T-3	What methods can I call on a Scheduled TimerTask?
N	T-4	What is the difference between schedule(TimerTask task, long delay, long period) and scheduleAtFixedRate(TimerTask task, long delay, long period)?
O	T-5	What state does a TimerTask need to be in to call scheduledExecutionTime?
C	T-6	Can I schedule a TimerTask that has already been scheduled?
N	R-1	How is a ResultSet instance created?
C	R-2	Can I call the getArray method when the cursor is on the insert row?
O	R-3	What state does the ResultSet need to be in to call the wasNull method?
T	R-4	How do I transition a ResultSet object from the ForwardOnly to the Scrollable State?
O	R-5	Which states does the ResultSet need to be in to call the updateInt method?
A	R-6	What state is the ResultSet object if a call to the next method returns false?
T	R-7	How do I transition a ResultSet object from the CurrentRow to the InsertRow state?
N	R-8	Why does getMetadata take no arguments and getArray take a int columnIndex or String columnLabel as an argument?
C	R-9	Can I call the isLast method on a forward only ResultSet?
A	R-10	What states could the ResultSet object in when a call to the next method throws a java.sql.SQLException because it is in the ResultSet is in the wrong state?
A	U-1	What state is the URLConnection in after successfully calling the getContent method?
C	U-2	If the URLConnection is in the connected state can I call the setDoInput method?
N	U-3	How do I create a URLConnection instance?
O	U-4	What state does the URLConnection need to be in to call the getInputStream method?
T	U-5	What method transitions the URLConnection from the Connected to the Disconnected state?

Category definitions

A Instance of the "What abstract state is an object in?" question category.

C Instance of the "What are the capabilities of an object in state X?" question category.

N Instance of the non-state question category.

T Instance of the "How do I transition from state X to state Y?" question category.

O Instance of the "In what state(s) can I do operation Z?" question category.

to disadvantage Javadoc unnecessarily, so we tried to make it as easy as possible for participants to perform the mapping from the prose description in the Javadoc to the state names in the glossary. In two cases there was no obvious name to give the state from the Javadoc. First, we called a URLConnection that has not yet connected "Disconnected," which is a word that appears neither in the Javadoc nor the Java source code. Second, we called a TimerTask that is unscheduled, "Virgin" even though this word never appears in the Javadoc. In this case we borrowed the word from the implementation code—the state of a TimerTask is encoded with an integer, and the integer constant used for an unscheduled TimerTask is called VIRGIN. Finally, we wrote all of the descriptions to succinctly explain the meaning of the state name.

All of the non-state questions require understanding a non-state detail of the API or comparing two details. Since the Plaiddoc API documentation is larger than the Javadoc documentation one might expect that it would be slightly easier to answer these questions with Javadoc. Two of the non-state question are instances of "how do I create an instance of class X?", two ask participants to compare two methods (in one case the methods were in different states), and one asks participants to understand non-state details of the behavior of an individual method.

Participants were instructed to "find the answer to each question in the documentation and tell the experimenter the answer as soon as you have found it." Whenever a participant answered a question for the first time, the experimenter asked,"is that your final answer?" Participants were limited to ten minutes per task. The experiment proceeded to the next task whenever a participant answered a question and confirmed it or the time limit was reached. Participants were not told whether their answer was correct and the experiment proceeded regardless of answer correctness.

5.5 Post-experiment interview

After completing the experiment participants were asked four questions to see how well they could map the state concepts we trained them about before the study (e.g. and-states, or-states, state hierarchy, impact of transitions on and-states) to the particular APIs they saw in the study. For example, we asked "What is an example of two ResultSet and-states?" Participants were also asked to rate their affinity to the documentation they used, and if they used Plaiddoc to compare Plaiddoc to Javadoc on a five point Likert scale. Then they were asked "Which documentation format that you learned about before the study—Javadoc, Plaiddoc, or UML state diagram—do you think would have been most helpful to complete this study?" Finally, some individuals were also asked additional questions about their task performance at the experimenter's discretion.

6 Results

In this section, we discuss the study results and try to give the best evidence to answer the research questions presented in the introduction. We first compare the task completion performance of Plaiddoc and Javadoc participants. Then we compare the correctness of these responses provided by those same groups. We follow with an evaluation of the learning effects of performing study tasks. Finally we discuss the post-study interview and pilot results. Raw timing and correctness data is available on the web.[4]

6.1 Task Completion Time

In this subsection we discuss the results related to the task completion time output variable. This output variable addresses RQ1 and RQ2 (Can programmers answer state search questions more efficiently using Plaiddoc than Javadoc? and Are programmers as effective answering non-state questions using Plaiddoc as they are with Javadoc?) by comparing task completion times across conditions.

To determine completion time we analyzed the video and marked when we finished reading the task question and when the participant confirmed his or her "final answer." The difference between these two marks was noted in the task completion time.

The ten-minute task time limit was reached by many participants on question R-4, but never on any other question. In fact, only two participants exceeded five minutes while answering any other question, and they did so for only one question each. Timeouts are not directly comparable to other timing data, and therefore we evaluate question R-4 separately, and in detail, in §6.2. This subsection does not include data from question R-4.

The total completion time for each of the Plaiddoc and Javadoc participants on state questions is visualized by the box plot in Figure 2(a), and for non-state question in Figure 2(b). A two-factor fixed-effects ANOVA revealed no significant interaction between documentation type and task ordering (p=0.25) on total task completion time. Therefore, we compare all 10 Plaiddoc participants against their 10 Javadoc counterparts.

The mean total completion time of all state search tasks was 10.3 minutes in the Plaiddoc condition, and 22.4 minutes in the Javadoc condition (2.17x difference). An independent samples two-tailed t-test revealed that the difference is statistically significant ($p < 0.001$). The difference between the means was 12.1 minutes, and 95-percent confidence interval was 6.38 to 17.8 minutes.

The mean completion time of non-state tasks was 5.77 minutes in the Plaiddoc condition, and 5.95 minutes in the Javadoc condition. Unsurprisingly, this difference is not statistically significant (p=0.802). The 95-percent confidence interval of the difference is -1.32 to 1.68 minutes.

The four state search categories can be subdivided into two categories. In two of the search categories, a participant begins his or her search at a state and tries to find a method.[5] In the other two search categories the participant starts

[4] `http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/RawPlaiddocStudyData.pdf`

[5] What are the capabilities of an object in state X? How do I transition from state X to state Y?

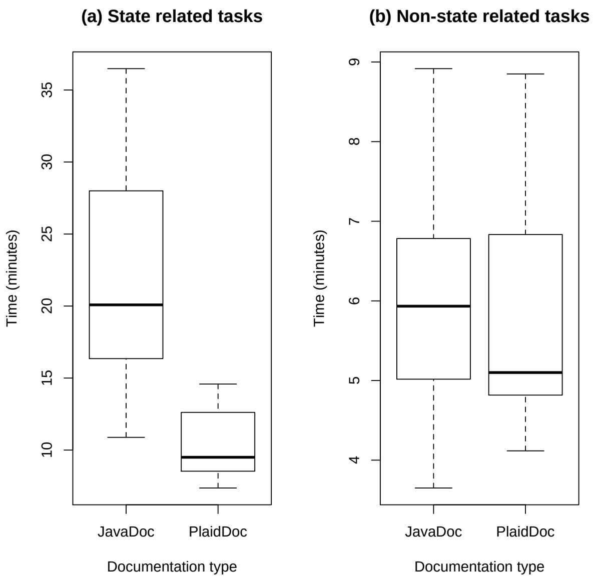


Fig. 2. Box plot comparing the completion time of Javadoc and Plaiddoc participants.

at a method or other detail (e.g. exception, instance creation), and tries to find a state.[6] Since methods are organized in Plaiddoc by state one would expect that Plaiddoc would improve performance primarily for searches that proceed from a state to a method. This hypothesis turns out to be correct — Plaiddoc outperformed Javadoc in these categories by 2.41x. However, one might expect that Plaiddoc would not be helpful in the method first categories, but Plaiddoc outperformed Javadoc by 1.87x in these categories. Therefore, Plaiddoc appears to be more helpful for state-first search than method-first search. We performed two factor, fixed-effects ANOVA in which the two factors are documentation type and search type and the output variable is time. The interaction term between documentation type and search type is only marginally significant (p=0.089).

Demographics. We did not balance participants in conditions by any demographic factor. By random chance, six of nine students with experience and three of four with more than one year of experience were assigned to the Javadoc conditions. However, experience had no significant impact on the timing results. A two-factor ANOVA where the two factors were experience and documentation type showed no significant effects from experience (F=.058, df=1, p=.813) or the experience by documentation type interaction term (F=1.34, df=1, p=.719).

[6] What abstract state is an object in? In what state(s) can I do operation Z?

Table 2. Correctness results for each participant on the 16 state search questions.

	Paricipant #																				Total	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Pdoc	Jdoc
DocType	P	P	J	J	P	P	J	J	P	P	J	J	P	P	J	J	P	P	J	J	P	J
Correct	15	15	14	16	15	16	15	14	15	15	14	14	15	15	16	16	15	15	11	13	151	143
Incorrect	1	0	2	0	0	0	1	1	0	0	2	2	1	0	0	0	0	0	5	2	2	15
Timed-out	0	1	0	0	1	0	0	1	1	1	0	0	0	1	0	0	1	1	0	1	7	2

Feature comparison discussion. Every participant used text-search (i.e. CTRL-F in the browser window) to find method names. They then used the location in a state box, pre-conditions, post-conditions, and state relationship diagrams to answer the question efficiently. Plaiddoc is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. The difference in relative performance between the state categories allows us to (very roughly) compare the benefits of state organization to the other three features. Since the method based search does not benefit from the state-based organization, all of the performance differences observed in the method based search tasks are likely to derive from explicit state transitions, state-based type specifications, and rich state relationships. The extra performance of the state based search is likely to derive from the state-based organization. We do not think it's possible to separate the benefits of the embedded state diagram from the preconditions and postconditions. In one early pilot we did not include the state diagram and the participant struggled to answer questions that required knowledge of state relationships. Similarly, a state diagram without detailed information about the requirements and impact of method calls would likely not be effective.

6.2 Correctness

Almost half of the participants provided at least one wrong "final" answer to a state-search question. Among the 320 total answers provided to the 16 state search questions 294 were correct, 17 incorrect, and nine were not provided because the question timed out. In this subsection, we compare the correctness of Plaiddoc answers to Javadoc answers (RQ3). The number of right, wrong, and timed-out answers for each participant are shown in Table 2.

Only two of the 17 wrong answers were provided by Plaiddoc participants. Plaiddoc participants answered 98.75% of the questions correctly, and Javadoc participants answered 90.5% correctly. The odds ratio in the sample is 7.92.[7] We analyzed the contingency table of Javadoc vs. Plaiddoc and Correct vs. Incorrect using a two-tailed Fisher's exact test. The contingency table is shown in Table 2 in the rows labeled "Correct" and "Incorrect" and the columns labeled "Pdoc" and "Jdoc". The test revealed that the difference is very significant (p=0.002). The 95-percent confidence interval of the odds ratio is 1.78 to 72.1.

[7] The odds ratio is a standard metric for quantifying association between two properties. In our example, it is the ratio of the odds of being correct when using Plaiddoc to the odds of being correct when using Javadoc.

Incorrect responses. All of the wrong answers and time-outs were provided to just five of the 16 state questions. No wrong answers were provided to any of the non-state questions. It is worth discussing the content of the wrong answers to provide insight into the types of problems programmers face when answering state-related questions.

In response to question T-3, a Plaiddoc participant (#19) incorrectly suggested that none of the TimerTask methods could be called on a scheduled TimerTask because "the methods are called by the Timer." This participant correctly noted the main mode of usage, but incorrectly assumed this was the exclusive mode of usage.

In response to question T-5, three[8] Javadoc participants incorrectly suggested that TimerTask scheduledExecutionTime can be called in any state when in fact it can only be called in the executed state. Three of these wrong participants noted correctly that scheduledExecutionTime does not throw an exception. Unfortunately, not every protocol violation results in an exception, a fact that was noted in pre-test training.[9] In this case, the protocol is documented in the description of the return value, which is described as "undefined if the task has yet to commence its first execution." In the post-experiment interview all three incorrect participants said that they did not notice this return value description.

In response to T-6, two Javadoc participants incorrectly replied that one can schedule an already-scheduled TimerTask. Participant #19 answered very quickly (15 seconds) without thoroughly examining the documentation. Participant #8 read aloud from the documentation, noting that the method throws an IllegalStateException "if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated." However, #8 somehow skipped "scheduled or" while reading.

Three Javadoc participants and one Plaiddoc participant incorrectly answered U-5. The question asks, "What method transitions the URLConnection from the Connected to the Disconnected state?" There is no such method, as 16 participants correctly noted. The three incorrect Javadoc participants suggested one could transition the URLConnection to the Disconnected state by calling its setConnectionTimeout method with 0 as the timeout value argument. This method "sets a timeout value, to be used when opening a communications link to the resource referenced by this URLConnection. If the timeout expires before the connection can be established, a java.net.SocketTimeOutException is raised." Therefore, setConnectionTimeout has no impact at all on a URLConnection instance that has already connected. Participant #1, a Plaiddoc participant, incorrectly answered that the non-existent "disconnect" method could be used to transition the URLConnection. This was the last question that participant

[8] Participant #19 also answered T-5 incorrectly because, as in question T-3, #19 thought all TimerTask "methods are called by the Timer" including scheduledAtFixedRate.

[9] The openTrunk method's protocol is documented by its description of the return value Javadoc training materials.

#1 answered, so perhaps #1 was ready to leave and so didn't investigate this question thoroughly.

Finally, R-4 produced the most varied responses. The question asks the participant to transition a ResultSet object from the ForwardOnly to the Scrollable state. However, no transition is possible since ForwardOnly and Scrollable are *type qualifiers* and therefore are permanent after instance creation. Seven Plaiddoc and two Javadoc participants never answered this question because they timed out. One Plaiddoc and five Javadoc participants answered the question incorrectly. Many of the timed-out Plaiddoc participants considered but then ultimately rejected the incorrect answers provided by the Javadoc respondents. This suggests that the specifications provided by Plaiddoc participants can provide confidence that an answer is *incorrect*. The Plaiddoc participants likely traded no-answers for incorrect answers.

Four Javadoc participants incorrectly answered that the setFetchDirection method will transition a ResultSet object from the ForwardOnly to the Scrollable state. Unfortunately, this method does no such thing, instead it "gives a hint as to the direction in which the rows in this ResultSet object will be processed." These four participants did skim the description, but it seems that they relied primarily on the method name to make their determination.

One Javadoc and one Plaiddoc participant noticed the following sentences in the class description: "A default ResultSet object is not updatable and has a cursor that moves forward only ... It is possible to produce ResultSet objects that are scrollable." which is immediately followed by a code example in which the createStatement method is called on TYPE_SCROLL_INSENSITIVE as an argument on a *connection* instance. Upon reading this, both participants immediately answered that the createStatement method should be called on a *ResultSet* instance. The Plaiddoc participant even suggested that the createStatement was missing from the method details list because "Plaiddoc is just a prototype."

Questions U-5 and R-4 both ask participants to find a method that does not exist. These questions, like all state-search questions in the study, are derived from the questions participants asked in the observational study discussed in Sunshine [31, ch.3]. However, participants in empirical studies are well-known to be compliant to experimenter demands. Therefore, some may therefore consider them to be "trick" questions. If these questions are excluded, then Plaiddoc participants answered 140 state-search questions correctly (100%) and 0 incorrectly while Javadoc participants answered 133 correctly (95%) and 7 incorrectly. A two-tailed Fisher's exact test of this contingency table is statistically significant (p=0.014). Since Plaiddoc participants in this sample answered every question correctly, the odds ratio is infinite. The 95-percent confidence interval of the odds ratio is 1.48 (the corresponding value is 1.78 when including every state-search question) to infinity (7.92 when including state-search question). Therefore, Plaiddoc participants were significantly more likely to respond correctly than Javadoc participants even when excluding "trick" questions.

Discussion. Three themes emerge from the incorrect and timed-out answers provided by participants. First, all of the time-outs occurred in question R-4 when participants were asked to find a non-existent method to transition between two states. Therefore, to answer this question correctly, participants needed to prove the absence of something to themselves.[10] Some participants felt the need to perform a brute force search of the method documentation to ensure that no methods were available that perfumed the transition. Of particular note, Plaiddoc participants didn't seem to trust that the ForwardOnly section of the Plaiddoc contained all of the potential methods.

It is also worth noting that question U-5 is in the same category but resulted in no time-outs. One possible explanation is that the ResultSet interface is much larger than the the URLConnection class,so it is easier to be confident that no such method exists. In addition, participants seemed to intuit that the URLConnection transition is missing, but not intuit that the ResultSet transition is missing.

Second, the questions required the participants to digest a lot of text. Participants commonly relied on heuristics and skimming to answer questions quickly. For example, the five Javadoc participants who answered R-4 with setFetchDirection matched the method name to the task and quickly confirmed the match in the description, but did not fully digest the description text. The participant who missed the word "scheduled" in the exception details was being similarly hasty. This phenomenon may partially explain why Plaiddoc participants were so much quicker than Javadoc participants, as we saw in §6.1. Plaiddoc presents a natural heuristic to participants — when examining a method, look first at the state it is defined in, then at its preconditions and postconditions.

Third, participants were tripped up by non-normal modes of use. We saw that participant #19 thought only the Timer could call TimerTask methods because that is the normal mode of use. Similarly, most protocol violations throw exceptions and are documented in the method or exception descriptions. However, scheduledExecutionTime somewhat abnormally documents the protocols in the return value description which confused three participants. Finally, abstract states normally map well to the primitive state of object instances. However, a URLConnection that has been disconnected from the remote resource is not in the Disconnected abstract state, as expected by three participants.

6.3 Learning

To answer RQ4, which asks whether state search performance improves with practice, we alternated the order that question batches were asked of participants. As we describe in §5.3, half of the participants first received URLConnection questions and half first received Timer questions. The output variable we discuss in this section is the ratio of total Timer batch completion time to total URLConnection batch completion time (the "T/U ratio"). If learning occurs,

[10] In [31, ch.3] many forum questioners had similar problems with missing state transitions.

Table 3. Analysis of observed variance of T/U Ratio. The fixed-effects sources of variation considered are documentation type and batch order.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
DocType	1	0.06695	0.06695	0.4560	0.50914
BatchOrder	1	0.96519	0.96519	6.5737	0.02081
DocType:BatchOrder	1	0.51496	0.51496	3.5073	0.07949

then the T/U ratio should be larger for participants who performed the Timer batch first than for those who performed the URLConnection batch first.

In the Javadoc condition, the mean T/U ratio of the Timer first sub-condition is 1.07 and .948 in the UrlConnection first sub-condition. This difference is not statistically significant (p=0.695). On the other hand, in the Plaiddoc condition the mean T/U ratio of the Timer first sub-condition is 1.50 and 0.743 in the UrlConnection first sub-condition. An independent samples two-tailed t-test shows that this difference is statistically significant (p=0.003).

We performed a two factor, fixed-effects ANOVA in which the two factors are documentation type and batch order and the output variable is the T/U ratio. The results are show in Table 3. This ANOVA reveals that there is a marginally significant interaction between documentation type and batch ordering (p=0.079). This should be interpreted as weak evidence that task-completion speed improved more for Plaiddoc participants than for Javadoc participants. However, more data is needed to know for sure.

Discussion. The Plaiddoc participants performance improved significantly during the study, which is perhaps unsurprising since Plaiddoc was new to all of the participants. We would like to say with confidence that state-search performance of programmers using Plaiddoc would improve over time relative to programmers using Javadoc. However, the learning observed in the Plaiddoc condition was not significantly stronger than the learning observed in the Javadoc condition.

6.4 State concept mapping

To investigate RQ5, we asked four questions to map the concepts they learned about in training to the Timer, TimerTask, ResultSet, and URLConnection. Plaiddoc participants responded correctly 23 of 40 times, while Javadoc participants answered correctly 25 times. This difference is not statistically significant.

Discussion. We hypothesized that Plaiddoc participant would be better at mapping API specifics to general state concepts. We thought this because Plaiddoc makes many state concepts more salient. There is no evidence for this hypothesis in the data. Javadoc participants spent much more total time with the documentation and they read much more of the detailed prose contained inside the documentation. Perhaps this extra time and detail compensated for the state salience of Plaiddoc.

We told all of the participants that timed out while trying to find a method to transition the ResultSet from ForwardOnly to the Scrollable state, that the method did not exist. We asked if they had any ideas about how to better represent missing state transitions. Most didn't give any suggestion, but one suggested that methods that perform state transitions should be separated from other methods so they're easier to find. This suggestion is worthy of further investigation.

6.5 Participant preference

In the post-experiment interview we also gauged participant preferences. Nine of ten Plaiddoc participants said that a different documentation format would have been more helpful in performing the study. Seven selected UML state diagrams and two selected Javadoc. The Javadoc participants also primarily selected UML State diagrams (five of ten), followed by Javadoc (3), and Plaiddoc (2).

Discussion. The results in this study show that Plaiddoc participants outperformed Javadoc participants. Therefore participant preferences does not match the measured outcome. Why do so many Plaiddoc participants prefer another documentation format? The simplest explanation is that Plaiddoc is unfamiliar, while Javadoc is familiar. In addition, one participant in the Plaiddoc condition who preferred Javadoc explained that he "felt lost" while using Plaiddoc. A Plaiddoc page is divided into many more subsections (one for each state) than a Javadoc page. Improved visual cues indicating the which state is being viewed might alleviate this problem. Another possible reason, is that the Plaiddoc state diagram is produced in ASCII and therefore looks old and amateurish. The state diagram does not match well with the modern look of the rest of the page. Regardless of the reason for the preference, this study's results are a cautionary tale for researchers who rely only on user preferences to evaluate tools.

7 Threats to Validity

In this section we discuss threats to validity of our causal claims. We divide this section using the canonical categories of validity: construct validity, internal validity, and external validity.

7.1 Construct validity

We trained all participants equally, including training of Javadoc participants to use Plaiddoc. There is some risk in this design that Javadoc participants will be disappointed that they did not get to use Plaiddoc. They were familiar with Javadoc so they may have preferred to try something new. Therefore, Javadoc participants may have performed worse because they experienced what Shadish [27, p. 80] calls "resentful demoralization." Two facts suggest that demoralization had at most a small effect on the results: First, only two of 10

Javadoc participants said they would have preferred to use Plaiddoc in the post-experiment interview. Second, both Javadoc and Plaiddoc are documentation formats and neither is particularly exciting. The classic examples in which "resentful demoralization" was measurable include much more severe differences between the control group and the experimental group. Fetterman [12] describes an experiment evaluating a job-training program in which the control group includes participants who were denied access to the training program. Walther [35] compared an experimental group that is paid a substantially higher participation reward to a control group paid much less. We would not expect to see anywhere near as much demoralization in our study as in these studies, even for participants who would have preferred to use Plaiddoc.

Although participants were never told explicitly, it is likely participants realized that Plaiddoc was our design. Therefore, Plaiddoc participants may have performed better and Javadoc participants worse because of "experimenter expectancies" [25, p. 224]. In other words, the very fact that we expected Plaiddoc to outperform Javadoc *and* the participants could possibly infer this expectation, may have impacted in the result in the direction we expected.

7.2 Internal validity

The focus of this study's design is internal validity. Participants were randomly assigned, participants were isolated from outside events in equivalent settings, we used a between-subjects design, and there was no attrition during the study. All that being said, one threat to internal validity is worth mentioning. Participants were assigned to conditions randomly, but it could be that the participants in the Plaiddoc group were better equipped to answer the questions in the study. We discussed the distribution of programming experience in §6.1 and showed that it did not seem to have an effect on outcomes. However, it could be the groups differ along another dimension—for example, programming skill, experience with protocols, intelligence—that we did not measure and this impacted the results.

7.3 External Validity

Our earlier qualitative studies and the experiment discussed here have opposing strengths and weaknesses. The qualitative studies emphasize external validity with realistic tasks and professional participants, but cannot be used to draw conclusions about causal relationships. The experiment in this paper focuses on internal validity with a carefully controlled experimental design that allows strong causal conclusions. However, the external validity of the experiment is enhanced because participants performed tasks in which they were required to tackle protocol programming barriers observed in the qualitative studies. Therefore, the experimental results are likely to translate to real-world problems and the processes that programmers use to solve them. All that being said, the threats to external validity in those earlier studies extend into this study [31, §3.4].

The state search tasks are connected to our qualitative results—they use the same APIs that were problematic for Stack Overflow questioners and they are instances of the state search categories that were observed repeatedly in the observational study. However, the non-state search tasks did not come from developer forums or any other real-world programming resource. Instead they were designed to simply *not* make use of Plaiddoc's novel state features. In our results, Plaiddoc participants did not perform worse on these tasks than Javadoc participants. However, it could be that there are other important categories of tasks for which Javadoc is better than Plaiddoc.

Another noteworthy external validity concern in the experiment here has to do with the student population studied. None of the participants seem to have struggled with the concept of preconditions and postconditions which are used heavily by Plaiddoc. This may be because the concept as used in the study is simple, but it may also be that the Carnegie Mellon student population we studied is especially exposed to formal methods. The very first course in the Carnegie Mellon undergraduate computer science sequence teaches students to verify imperative programs with Hoare-style contracts.

8 Type annotations as documentation

Many research groups have developed specialized type-based annotation systems for particular domains. Prominent examples include information flow [26], thread usage policies [33], and application partitioning [7]. In the vast majority of these systems, including all of the examples just cited, the primary benefit of the annotation systems touted by their creators is either verification or automated code generation. The preconditions and postconditions that appear next to methods in Plaiddoc are essentially state-based type annotations. Therefore, this study provides indirect evidence that type based annotations can have benefits as documentation.

In the last few years, there have been a flurry of studies comparing the benefits of static and dynamic types [16, 29, 17]. This research suggests that dynamic types have an advantage for small, greenfield tasks, while static types have an advantage for larger, maintenance tasks.

The most closely related study [22], evaluated the benefits of type annotations in undocumented software. The results were mixed—types were significantly helpful in some tasks, and significantly harmful in others. One possible interpretation of the results is that types were helpful in tasks that were more complex (involved more classes) and harmful otherwise. Our results provide a clearer picture — Plaiddoc provided benefits in every state-search category. In their study, programmers performed programming tasks using two "structurally identical," synthetic, undocumented APIs. In our study, programmers answered search questions with well-documented real-world APIs. One important consequence of these differences, is that our study evaluates types *only* for their documentation purpose, while theirs evaluates the collective value of both static-checking and types as documentation.

9 Conclusion

In this study we demonstrate the effectiveness of Plaiddoc documentation relative to Javadoc documentation in answering state-related questions. The barrier to entry for using the Plaiddoc tool are minimal—only 1-3 annotations are required per method. We annotated all three APIs in less than one day of work. The main barrier to using Plaiddoc in production is training programmers to consume the documentation effectively. Untrained participants in pilot studies were not able to use Plaiddoc effectively. Even basic protocol concepts were foreign to our participants before training. That said, the training we provided was very quick and required no specialized knowledge. Regardless, it seems clear that any mainstream language that adopts first-class state constructs should also adopt a Plaiddoc like documentation structure. More generally, our study shows that state-based type annotations provide documentation-related benefits even for well-documented code. Thus, our results open the door to future work investigating the documentation-related productivity benefits of type-like annotations in a broad range of domains.

Acknowledgements. This work was supported by supported by the U.S. National Science Foundation under grants #CCF-1116907 and #IIS-1111750. National Security Agency lablet contract #H98230-14-C-0140, and the Air Force Research Laboratory.

References

1. N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *ECOOP 2011 – Object-Oriented Programming*, pages 2–26. Springer Berlin Heidelberg, 2011.
2. N. E. Beckman and A. V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 211–221, New York, NY, USA, 2011. ACM.
3. K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 195–219, Berlin, Heidelberg, 2009. Springer-Verlag.
4. M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 260–269, New York, NY, USA, 2010. ACM.
5. P. Chandler and J. Sweller. Cognitive load theory and the format of instruction. *Cognition and Instruction*, 8(4):293–332, 1991.
6. M. T. Chi, M. Bassok, M. W. Lewis, P. Reimann, and R. Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13(2):145 – 182, 1989.
7. S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-first ACM*

SIGOPS Symposium on Operating Systems Principles, SOSP '07, pages 31–44, New York, NY, USA, 2007. ACM.

8. G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Program abstractions for behaviour validation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 381–390, New York, NY, USA, 2011. ACM.
9. U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 320–330, 2009.
10. M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.
11. B. Ellis, J. Stylos, and B. Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 302–312, 2007.
12. D. M. Fetterman. Ibsen's baths: Reactivity and insensitivity. (a misapplication of the treatment-control design in a national evaluation). *Educational Evaluation and Policy Analysis*, 4(3):261–279, 1982.
13. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.
14. M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.
15. T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.
16. S. Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM.
17. S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, pages 1–48, 2013.
18. R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 237–240, New York, NY, USA, 2005. ACM.
19. C. N. Jaspan. *Proper Plugin Protocols.* PhD thesis, Carnegie Mellon University, December 2011. Technical Report: CMU-ISR-11-116.
20. B. E. John and D. E. Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3(4):320–351, Dec. 1996.
21. J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65 – 100, 1987.
22. C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 683–702. ACM, 2012.

23. L. R. Neal. A system for example-based programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '89, pages 63–68, New York, NY, USA, 1989. ACM.
24. M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013.
25. R. Rosenthal and R. L. Rosnow. *Essential of Behavioiural Research: Methods and Data Analysis*. McGraw-Hill Higher Education, New York, NY, USA, third edition, 2008.
26. A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.
27. W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Wadsworth Cengage Learning, 2002.
28. J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On breaking SAML: Be whoever you want to be. In *Proceedings of the 21st USENIX conference on Security symposium, Security*, volume 12, pages 21–21, 2012.
29. A. Stuchlik and S. Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 97–106, New York, NY, USA, 2011. ACM.
30. J. Stylos and B. A. Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 105–112, New York, NY, USA, 2008. ACM.
31. J. Sunshine. *Protocol Programmability*. PhD thesis, Carnegie Mellon University, December 2013.
32. J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class state change in plaid. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM.
33. D. F. Sutherland and W. L. Scherlis. Composable thread coloring. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 233–244, New York, NY, USA, 2010. ACM.
34. B. Ullmer and H. Ishii. Emerging frameworks for tangible user interfaces. *IBM Systems Journal*, 39(3.4):915–931, 2000.
35. B. J. Walther and A. S. Ross. The effect on behavior of being in a control group. *Basic and Applied Social Psychology*, 3(4):259–266, 1982.
36. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 218–228, New York, NY, USA, 2002. ACM.
37. Y. Ye, G. Fischer, and B. Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 60–68, New York, NY, USA, 2000. ACM.
38. H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.

Searching the State Space: A Qualitative Study of API Protocol Usability

Joshua Sushine, James D. Herbsleb, and Jonathan Aldrich
{sunshine, jdh, aldrich}@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA

***Abstract*—Application Programming Interfaces (APIs) often define protocols — restrictions on the order of client calls to API methods. API protocols are common and difficult to use, which has generated tremendous research effort in alternative specification, implementation, and verification techniques. However, little is understood about the barriers programmers face when using these APIs, and therefore the research effort may be misdirected.**

To understand these barriers better, we perform a two-part qualitative study. First, we study developer forums to identify problems that developers have with protocols. Second, we perform a think-aloud observational study, in which we systematically observe professional programmers struggle with these same problems to get more detail on the nature of their struggles and how they use available resources. In our observations, programmer time was spent primarily on four types of searches of the protocol state space. These observations suggest protocol-targeted tools, languages, and verification techniques will be most effective if they enable programmers to efficiently perform state search.

I. Introduction

Application Programming Interfaces (APIs) often define *protocols* — restrictions on the order of client calls to API methods. These protocols are common: more than three times as many types in the Java Standard Library define protocols as define type parameters [2]. Protocols can also be complex: ResultSet from the Java database connectivity (JDBC) library contains 33 unique states dealing with different combinations of openness, direction, random access, and insertions [3]. Protocols also cause significant pain: for instance, in a study of problems developers experienced when using a portion of the ASP.NET framework, three quarters of the issues identified involved temporal constraints such as the state of the framework in various callback functions [16]. Finally, protocols are poorly supported by mainstream languages and tooling — the state of practice is to specify protocols with documentation, implement them with low-level language constructs, and react to violations with exceptions.

All of the factors just mentioned have spurred a tremendous number of research projects aimed at improving the usability of API protocols. There have been many tools and languages designed to specify and verify protocols. Strom and Yemini [28] proposed *typestate* as a compiler checkable abstraction of the states of a data structure. The Fugue system later integrated typestates into an object-oriented programming language [8]. Many tools verify protocols (e.g. Bierhoff et al. [4], Dwyer et al. [9], Foster et al. [11]). These tools require programmers to specify protocols using alias and typestate annotations separate from code. To automate the annotation, many tools mine protocol specifications from program executions or static analysis. A recent survey of automated API property inference techniques uncovered 35 inference techniques for ordering specificationss [25].

This massive research effort has gone on despite the fact that very little is known about precisely what problems programmers have when using APIs with protocols. In this work we attempt to answer four research questions which we hope will provide more solid guidance for future researchers:

RQ1 What are the characteristics of protocol tasks that are difficult for programmers?
RQ2 How do programmers approach protocol tasks?
RQ3 What information do programmers seek and have difficulty locating while performing protocol tasks?
RQ4 What resources do programmers use while performing protocol tasks?

To answer these questions, we performed two studies of professional developers. The first study identifies real-world phenomena, and the second investigates the heart of those phenomena in more detail.

In the first study, we searched the popular developer forum, Stack Overflow, for questions related to known APIs with protocols. We then winnowed, analyzed, distilled, and merged the resulting questions into a list of distinct protocol-specific tasks. These tasks represent real protocol programming challenges and we noted five common characteristics in answer to RQ1.

In the second study, we brought seasoned professional programmers into the lab and observed them performing the tasks uncovered by the forum mining. To answer RQ2, we analyzed the transcripts to categorize the activities that programmers performed. Information seeking dominated programmer effort and we therefore noted the information the developers sought while performing the tasks and how they sought it. We found that developer time was spent primarily on state search. We also found that developers debugging protocol violations looked first to the documentation related to the method call occurring at the exception location to solve their problems. These findings address RQ3 and RQ4.

II. Protocols

This paper intends to investigate API protocols, but the term protocol is widely used with conflicting or ambiguous definitions. In this paper, we focus on object-oriented APIs and we borrow the precise definition from Beckman et al. [2, p. 4]:

> A type defines an *object protocol* if the concrete state of objects of that type can be abstracted into a finite number of abstract states of which clients must be aware in order to use that type correctly, and among which object instances will dynamically transition.

The focus of this definition is on state machines. An object with a protocol must have a finite number of states which are abstractions over concrete internal representations. These states are visible and relevant to API clients. An object transitions between abstract states when particular methods are called at runtime. Clients programs that do not comply with a protocol will cause the API to throw an exception, operate incorrectly, or fail to operate at all. We had this definition in mind while conducting both of the studies discussed in this paper. Therefore, all of the API protocols we studied conform to this state-based definition.

III. Related work

The studies we discuss in this paper focus on the usability of API protocols. This works builds on many recent studies of more general programming obstacles. Two classes of studies have particular relevance to this paper. The first class, which we will refer to as *information needs studies*, includes mostly-qualitative studies that are often conducted in the field. They investigate what information developers look for in their work, how they look for it, and the purpose of the information. The second class of studies, which we will refer to as *API usability studies*, are mostly quantitive and are usually conducted in the laboratory. They investigate the usability of particular APIs, or more recently the usability of API design choices. There is not space to discuss all of the examples in either class. Instead we will delve deeply into a few examples in each class (and one gap-bridger) to highlight important lessons for this paper and motivate the study's design.

A. Information needs studies

In an oft-cited example of an information needs study, Ko et al. [18] observed 17 Microsoft developers as they performed their regular work. During the study, participants searched for information 334 times, which the experimenters abstracted into 21 categories. The abstracted categories are all very high-level, reflecting the breadth of the activities performed. For example, the most common category was "did I make any mistakes in my new code?" Two categories identified by Ko are particularly relevant to protocols: 1) "What code causes this program state?" — A programmer using an API protocol needs to understand how an object transitions to a particular abstract state.[1] 2) "In what situations does this failure occur?" — Debugging a protocol violations requires understanding when a particular method call is invalid. Our studies expand on these results, discovering in more detail when question like these arise and what kinds of state information are needed.

Other studies, also information needs studies, have narrowed the developer tasks slightly to delve more deeply into specific topics. For example, Sillito et al. [26], like Ko, studied professional programmers in their work environments. However, instead of studying whatever the programmers happened to be working on, the programmers were asked to select an "involved" software change task, and never "a simple fix." Silito observed programmers pursued an answer to a higher-level question "by asking a number of other, lower-level questions." Sometimes the programmers even asked the low-level question first and built up to the higher-level question. Our studies will investigate which low-level questions are most useful in learning to correctly use API protocols.

LaToza et al. [20] brought programmers to the lab and asked them to contribute architecture-level design improvements to a 54KLOC open source tool. They noted several high-level differences between experts and novice participants: novices focused more on symptoms of problems, experts on sources; novices spoke in terms of specifics, and experts in terms of abstractions; novices wasted more time understanding implementation details, while experts' focus was wider. Again, these results are interesting and contribute to our general knowledge, but are of little direct utility to most language and tool designers.

Robillard and DeLine [24] surveyed and interviewed Microsoft developers about the obstacles they faced when they last learned to use a public API. The most common obstacles mentioned involved documentation. More particularly, the answers suggested five problematic issues commonly found in API documentation: design intent, code examples, matching APIs with use cases, penetrability, and formatting/presentation. Many of these issues were simply missing from documentation, (e.g. no discussion of performance characteristics), mistargeted (e.g. examples of inapplicable usage), or buried (e.g. most method documentation contains boilerplate repetition of information contained in the method signature).

B. API usability studies

The more traditional API usability studies observe programmers in the laboratory while they use APIs. In most of these studies, the participants performed tasks that were selected by the experimenters as "representative of typical use" of the API. McLellan et al. [22] were among the first to publish a study of a particular API, and they are also credited with spreading the recognition that "the techniques and theory developed for usability should be applied directly to the API" [6]. McLellan's study uncovered many low-level difficulties with the API under investigation, but more importantly for the purpose here, agreed with Robillard about the importance of code examples and documentation.

McLellan's study and those like it are primarily useful for the designers of the API under investigation. To provide guidance to

[1] Ko addressed this category with the Whyline tool [17].

designers of future APIs, Jeff Stylos and colleagues performed a series of studies to evaluate API "design choices" [10, 29, 30]. In Ellis et al. [10], the experimenters compared the usability of constructor-based instance creation with instance creation using a factory method or abstract factory [12], which Ellis refers to collectively as the "factory pattern." The study used both within and between subjects comparisons and found that users required much more time to instantiate objects when the API used the factory pattern rather than constructors.

The design choice studies provide data-driven design guidance, but it is difficult to abstract principles from them. For example, the Ellis study does not provide insight into why it is harder to use the factory method pattern than a constructor.

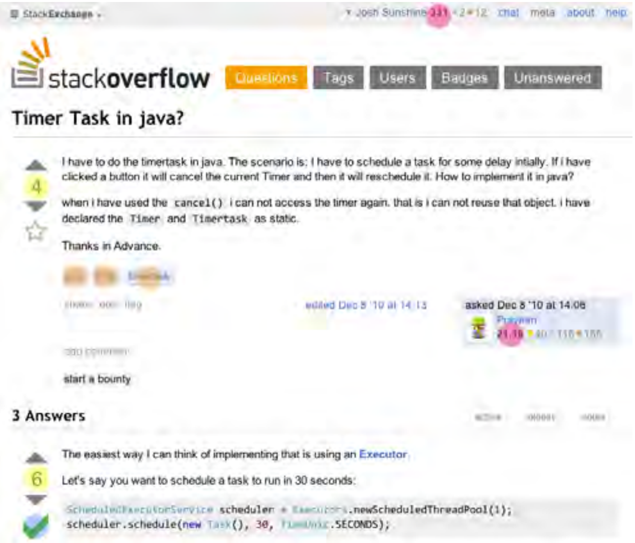
C. Discussion

The two studies we report in this paper lie between the two classes discussed above. The studies in this paper, like those in the first class, are qualitative and focus on the information needs of developers. Unlike the other information needs studies, we focus on a particular programming domain — API protocols— to add detail and richness to our existing general knowledge so that it can be used for tool building.

Our think-aloud laboratory study shares many elements with the studies in the second class. However, our tasks were mined from developer forums and we therefore expect the study to be more connected to practice. Finally, the laboratory study was not looking for quantitative results like the design-choices studies, nor specific issues with the APIs like the McLellan-type studies. Instead, the results of the second study are principles and understanding which we hope can be applied to any API with protocols. Our follow-on work, which validates this paper's conclusions, evaluates state-structured documentation using programming experiments that are similar to the McLellan-type studies [32].

IV. Forum Mining

We mined Stack Overflow, a widely-used developer forum, primarily to identify the characteristics of protocol tasks that are difficult for programmers (RQ1). We discuss the strengths and weaknesses of StackOverflow data in Section IV-A. We downloaded the entire Stack Overflow database which is freely available to anyone under a Creative Commons license. When this study was conducted there were 2.6 million questions on Stack Overflow. This is far too many to read and digest, so we winnowed the question list with the techniques we discuss in Section IV-B. The goal of the filtering was to focus our efforts on questions that were likely to be protocol-related and significant. Once we had a reasonable-sized list of questions, we manually read each questions to: 1) determine if the question was protocol-related, 2) distill a task, and 3) merge with existing tasks. This study's aim is to characterize recurring protocol problems, but does *not* attempt to estimate commonality. The study also required a lot of manual labor, so we likely excluded many protocol question for the sake of efficiency. The strategies we used in all of these efforts are discussed in Section IV-C. The most frequent and interesting characteristics of protocol-related questions are discussed in Section IV-D.



Timer Task in java?

I have to do the timertask in java. The scenario is: I have to schedule a task for some delay intially. If i have clicked a button it will cancel the current Timer and then it will reschedule it. How to implement it in java?

when i have used the `cancel()` i can not access the timer again. that is i can not reuse that object. i have declared the `Timer` and `Timertask` as static.

Thanks in Advance.

3 Answers

The easiest way I can think of implementing that is using an Executor

Let's say you want to schedule a task to run in 30 seconds:

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
scheduler.schedule(new Task(), 30, TimeUnit.SECONDS);
```

Fig. 1. Screen snap of the StackOverflow question page.

A. Strengths and weaknesses of forum data

Forums provide a window into developer practice that is particularly well suited to mining examples. Asking a question on a forum requires significant effort — it requires composing a question, extracting relevant code or documentation, and describing important context. After asking a question, the answers do not come immediately, so developers often wait to post questions until they have struggled for a while. Therefore, the questions usually contain distilled problems of practical significance.

We chose to use Stack Overflow for its wide use, feature set, and openness. Stack Overflow is the most popular developer forum on the web and it therefore contains questions in a uniquely broad set of categories. Parnin and Treude [23] found that StackOverflow covered 84% of the methods in the JQuery API. This was important for us because it allowed us to distill a wide-range of protocol-related tasks. A sample question page with important highlighted features is shown in Figure 1.

According to Mamykina et al. [21] Stack Overflow is also the fastest forum on the web, with median answer time of only 11 minutes. This speed encourages posting on low-level topics, which includes most protocol issues, since questioners can expect a fast answer. Mamykina credits the popularity primarily to the engagement of the Stack Overflow designers with the user community. In addition, the feature set, which includes a "reputation score" earned for asking well-liked questions or providing well-liked answers, incentivizes use [33]. All viewers of a question can categorize the question with a "tag," which helps programmers determine question relevance. Of particular importance to this effort is that questioners are rewarded for "accepting" an answer, which often gives the most important clue about the real problem the questioner faces. For example, the code search and recommendation tool Example Overflow uses these social features to the determine quality and relevance of programming examples contained in StackOverflow questions. [36].

Despite the numerous benefits of forum questions as a data source, and Stack Overflow in particular, the questions there are by no means representative of all programming problems. Vasilescu et al. [34] found that women are substantially less likely to participate in Stack Overflow than men. Furthermore, women that did participate were less likely to participate heavily or earn reputation points. More generally, Kuk [19] found that forum participants act strategically in a number of ways including by helping those who are likely to reciprocate and by seeking out career advancement opportunities. This strategic behavior results in a question and answer pool that is largely authored by a heavily active elite. Finally, the quality and difficulty of StackOverflow questions vary dramatically [13]. Therefore, one cannot count questions of a certain type to gauge commonality of that type. In summary, Stack Overflow is a useful resource for finding real-world programming problems but the participant and question population is not representative, nor are the questions sets directly comparable.

B. Winnowing the Question List

We wanted tasks that both are protocol-related and caused problems for real developers. Therefore, we started by assembling a list of 109 Java Standard Library classes that contain a protocol. The bulk of the classes are listed in two studies, Beckman et al. [2] and Whaley et al. [35], that identified protocols via semi-automated static analysis. Neither Beckman nor Whaley identified any protocols in interfaces, so 9 interfaces were added from other sources (e.g. Bierhoff et al. [4]). These interfaces are not implemented in the Java Standard Library, but they are implemented by many third parties, and so the interface protocols can be very widely used.

We downloaded a data dump from Stack Overflow that contained questions and answers that were created through March 2012.[2] We discarded 40 of the classes because their protocols were very familiar and simple. In particular three protocol patterns were removed: 1) Boundary protocols in which a method named next or starting with next (e.g. nextInt) cannot be called after the end of an underlying list (e.g. java.util.Iterator). 2) Deactivation protocols in which many methods cannot be called after the close method is called (e.g. java.util.Scanner). 3) Redundancy prevention protocols in which the cause of a Throwable or Exception cannot be set more than once[3]

We then searched for questions about each of 69 remaining classes systematically, to ensure that later analysis was done fairly. The SQL queries used are described in detail in Sunshine [31].

For most classes the search returned fewer than five related questions, and only nine had more than 100. In order to include only well-used APIs in the results, we focused our efforts on these nine classes.[4] We examined all of the questions and answers related to these nine classes, looking for protocol-related questions. We discuss how we determine if a question is protocol related in detail in Section IV-C. Of the nine, five had protocol-related questions: URLConnection, its close cousin HttpURLConnection, Timer, ResultSet, and Socket had protocol-related questions. The results in this section are drawn from questions related to these classes.

C. Analyzing a Question

We manually examined a total of 5,039 questions related to nine classes. The first order of business was to eliminate questions that were unrelated to protocols. The single fastest heuristic we used was to examine how the search keyword was used in the post. The keyword was often found in an import statement, method return type, type of an unused variable or argument, comment, throwaway reference, etc. but never used again. This phenomenon was especially common in cases where long code blocks were attached to a question for context. The vast majority (more than 90%) of questions were discarded by this heuristic alone. For example, in question #5302656, "java.sql.ResultSet" appears exactly once in a list of possible types of values accepted by a particular value. In question #2609535, ResultSet only appears in an import statement and is never used.

If the keyword heuristic did not eliminate a question, we examined the question more thoroughly. We focused on the accepted answers to questions, the exception types and error messages described, and searched all text and code for protocol violating methods. More details can be found in Sunshine [31, ch. 3].

Excluding questions. If none of the protocol violating methods appeared and none of the earlier strategies were useful, then we excluded the question from the study. It is therefore possible we incorrectly excluded questions this way, especially if the protocol issue was not in code but buried in difficult to parse prose. However, the large number of questions required us to be expedient. The goal of the study was not to estimate the commonality of protocol problems, but to characterize recurring patterns—which justifies the expediency.

Brute force. In rare instances, none of the above strategies worked. These instances usually included large blocks of code with many method calls and exceptions. When none of the earlier strategies worked, we carefully read the full text of the post, including all the answers, to understand the problem or problems faced by the questioner.

Distillation. If a question was found to be protocol related, we then distilled a concrete protocol-based task from the

[2] This was the latest data dump available at the time this part of the study was conducted.

[3] In unpublished experiments conducted by Ciera Japan, tasks involving these protocol patterns were very simple for expert developers, but still challenging for novices. It seems that experts have memorized or otherwise internalized the steps needed to use these libraries correctly. In these experiments, experts completed tasks involving these patterns very quickly and the observations therefore yielded little insight.

[4] We cutoff questions with fewer than 100 questions because 100 is a round number and there was a sizable gap in the data at that point. All of the APIs included in the final study had a minimum of 210 questions.

question being asked. We focused our efforts on discovering the particular difficulty the programmer had with the protocol. Protocols are composed of rules, and in most cases, the programmer violated one of these rules. In these cases, the distillation involved identifying the specific rule that was violated. We excluded all domain specific information from the task. For example a Timer running on Android is the same as a Timer running on a PC.

D. Results

After completing the winnowing, analysis, and distillation we selected 28 Stack Overflow questions. We merged these 28 question into 13 distinct topics. The results are summarized in Table I. The most common distilled question was about the violation of a protocol rule. There were 23 such questions and these were merged into nine topics, one for each distinct protocol violation (marked "Cannot" in the table).

Three questioners confused two rules that compose the protocol. These three questions represent two distinct confusions and they were therefore merged into topics (marked "Confusion" in the table). Finally, two questioners requested the APIs add a new protocol-related feature. These were distinct and therefore represent two topics (marked "Wanted" in the table). In both cases, the questioners requested state-tests, which we will discuss further in the next section. All of the questions, except in two topics, asked for help debugging a protocol violation.

1) Characteristics: The questions and corresponding topics had five common and interesting characteristics that we highlight here. These characteristics address RQ1, "what are the characteristics of protocol tasks that are difficult for programmers?" In each case we discuss the evidence for each characteristic in the data and then discuss its significance. After all the characteristics are introduced, we discuss the significance of the full collection.

Missing state transition. Many questioners hoped for or assumed a state transition that the protocol did not allow. For example, questioner #4278917 explicitly asks if there is a method that allows a client to "disconnect" and thereby reuse a URLConnection (there is none). Similarly, one way of looking at all six questions about rescheduling a TimerTask, is as a question about the ability to transition the TimerTask from the scheduled to the virgin state. Finally, two of the questioners trying to call scrolling methods on a forward-only specifically looked for a method to transition that ResultSet to the scrolling state. Documentation is particularly ill-suited to addressing this type of question. It often requires a global search of all of the method and class documentation to discover that a transition is not available.

State tests. For three of the four libraries, questioners asked for a method to test the abstract state of the object. The state test questions for Timer (#13880202) and URLConnection (#7614408) are listed in Table I. In addition, questioner #2741276 requests a method to test if a ResultSet has been closed. However, this question was not included in the results because an isClosed method was added in Java 6. Presumably, the questioner was using an earlier version of Java. There were no similar questions about Socket, but for good reason — Socket includes state tests for every state it defines.

State independence. In some cases, objects with protocols can occupy multiple states simultaneously. For example, a ResultSet object, whose UML state machine is shown in Figure 2, occupies the *and-states* Direction and Position simultaneously. State transitions on and-states act independently, and this independence confused several questioners. For example, the connectedness and openness of a socket are independent. Questioner #3701073, perhaps unsurprisingly, thought that a closed socket could not be connected, but this is incorrect. Similarly, the four forward-only questioners did not seem to understand that the act of calling a scrolling method did not change the Direction state.

Multi-object protocols. All four of the APIs we looked at closely inspired questions about the relationship to other APIs. For example, a ResultSet object is closed if the Statement object that created it is closed or reused. Four questioners in the sample struggled with this one issue (4646561, 4864920, 5840866, 10118129). Questioners also asked about the following other relationships: Timers with threads, Sockets with data streams, and URLConnections with Sockets. We did not include these multi-object protocol issues in the primary results to focus on the vast majority of protocol-specific tooling that does not support multi-object protocols.

Terminology Confusion. Many of the questioners seem to be confused by terminology. This type of confusion is extremely common and not protocol-specific. However, the frequency of its appearance in the data warrants a brief discussion. Questioners often assumed a particular definition for a term, and when the definition was wrong they struggled. For example, questioner #9497100 assumed that canceling a TimerTask would *always* abort the Task. The questioner therefore tried to cancel the task in the task's own run method, in a failed attempt to halt execution immediately. Other questions misinterpreted Socket.isConnected, Timer.schedule, and URLConnection.inputStream.

Discussion. All of the characteristics just highlighted, except terminology confusion, are protocol-specific. This suggests that protocol-targeted tooling or languages may be necessary to improve the usability of API protocols.

The challenge of missing state transitions suggests that documentation should include a list of state transitions in an easily digestible form. This would enable programmers to quickly learn which transitions are, and are not, available. The very existence of state test questions suggests the usefulness of state tests. Josh Bloch, the designer of much of the Java Standard Library including several of these classes, suggests that all APIs with protocols "should generally have a state-testing method indicating whether it is appropriate to invoke state-dependent method[s]." [5, p. 242].

That repeating occurrence of multi-object protocols in the forum mining data buttresses the evidence collected by Jaspan and Aldrich [14] that multi-object protocols are important. Therefore, this study motivates the those working on relationship types [15, 1]. Unfortunately, many protocol-targeted tools

TABLE I
LISTS THE APIS, QUESTIONS AND MERGED TOPICS DISCOVERED IN THE FORUM MINING.

API	Topic	#Qs	Question IDs
URLConnection	Cannot: Set request property after connected	2	331538, 5368535
	Cannot: Reuse connection	1	4278917
	Wanted: IsConnected state test	1	7614408
Timer	Cannot: Reschedule TimerTask	6	1041675, 1801324, 4388353, 6813654, 7631542, 8404736
	Cannot: Change Scheduled time of TimerTask	4	5014132, 6555583, 6762099, 8173147
	Confusion: Timer.cancel() vs. TimerTask.cancel()	2	1801324, 6477608
	Cannot: Cancel running TimerTask	1	9497100
	Wanted: State Test for TimerTask	1	13880202[a]
Socket	Confusion: Closed vs. Connected	1	3701073[b]
ResultSet	Cannot: Read after end	1	3502005
	Cannot: Call next on InsertRow	3	4874574, 6684753, 9836972
	Cannot: Call scrolling methods on forwardonly	4	6367737, 6871641, 8032214, 9007051
	Cannot: Read before calling next()	1	8039233

[a] This question was discovered after the forum mining, but matches all of the criteria used to select the other questions.
[b] This is the only Socket protocol question, but as of Sep. 2013 it had the highest reputation score in this table, suggesting its importance.

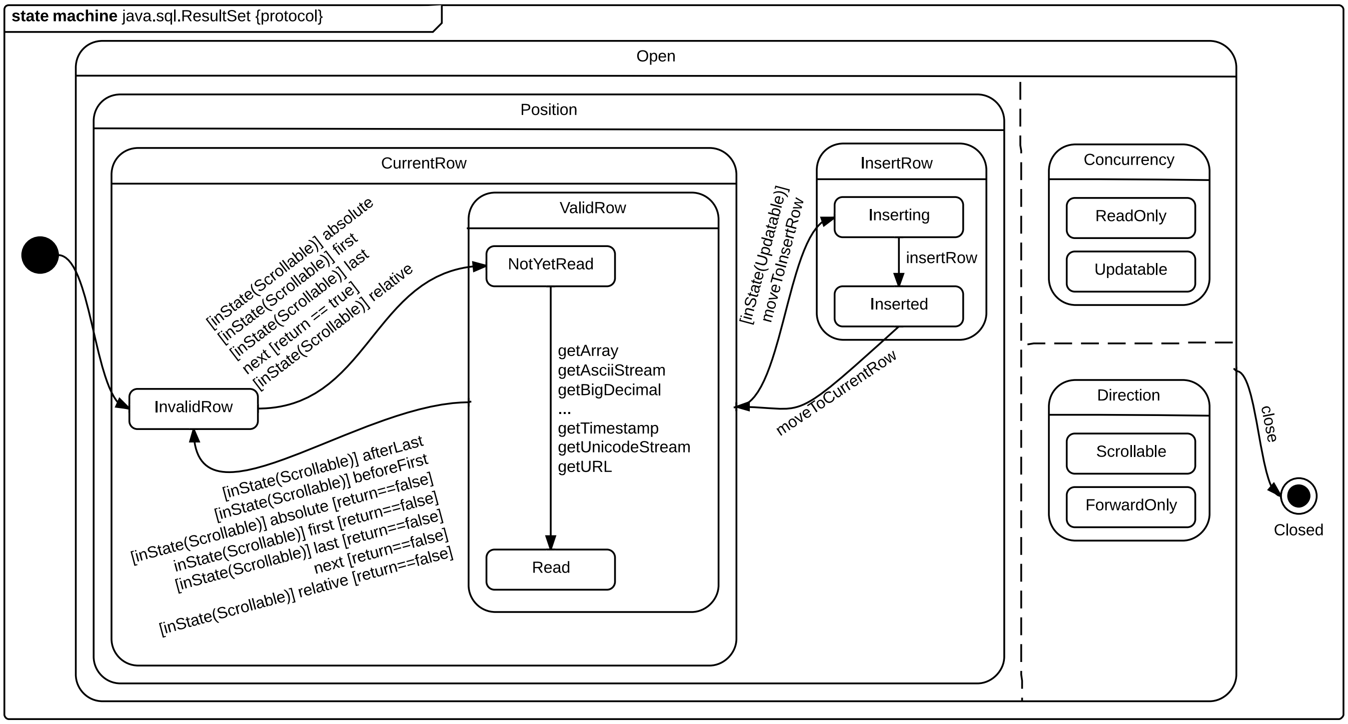

state machine java.sql.ResultSet {protocol}
Open
Position
CurrentRow
ValidRow
NotYetRead
[inState(Scrollable)] absolute
[inState(Scrollable)] first
[inState(Scrollable)] last
next [return == true]
[inState(Scrollable)] relative
InvalidRow
getArray
getAsciiStream
getBigDecimal
...
getTimestamp
getUnicodeStream
getURL
Read
[inState(Scrollable)] afterLast
[inState(Scrollable)] beforeFirst
[inState(Scrollable)] absolute [return==false]
[inState(Scrollable)] first [return==false]
[inState(Scrollable)] last [return==false]
next [return==false]
[inState(Scrollable)] relative [return==false]
[inState(Updatable)]
moveToInsertRow
InsertRow
Inserting
insertRow
Inserted
moveToCurrentRow
Concurrency
ReadOnly
Updatable
Direction
Scrollable
ForwardOnly
close
Closed

Fig. 2. UML State Machine for ResultSet.

do not support and-states. The data suggests and-states are particularly problematic, which in turn suggests that these tools are missing an opportunity to address an important usability challenge. Finally, the prevalence of terminology confusion, suggests that API protocol designers should carefully name state-related methods to ensure that the standard English definition matches its use in the protocol.

These characteristics share one significant weakness with the source from which they were derived. Each forum post represents a snapshot of a single programmer's thinking. It is difficult to know whether these characteristic problems are challenging for most programmers or just a tiny minority. Similarly, it is difficult to know what common programmer challenges were missed because they were resolved before a question was ever asked. Finally, and most significantly, the forum mining has given us a better idea of what is hard, but we still need to understand why they are hard. What do programmers do when trying to address these tasks? Why are their tools and documentations inadequate? We address these weakness in the laboratory observations we discuss next.

V. LABORATORY OBSERVATIONS

In this section, we describe the methodology and results of the laboratory study. The aim of this study is to learn how programmers approach protocol tasks (RQ2), with particular focus on the information they seek (RQ3) and the resources they use (RQ4). In this study, the tasks are taken from the forum mining and therefore connected to practice. We discuss how we transform the topics mined from Stack Overflow into tasks in the next section. We then discuss the study design. Next, we highlight observations from one particular task—inserting a new row into a ResultSet—which we will use to illustrate the important results from this study. Finally, we summarize the results from all of the tasks including quantitative and qualitative analysis.

A. Methodology

1) Topics to tasks: We converted each of the topics uncovered by the forum mining study, as summarized in Table I, into a corresponding programming task. The tasks were derived from the code contained in the topical question(s). The tasks did not include project context such as package names, or code that was not protocol related. Each task included instructions and a method annotated with pre and post-conditions. The source files are available on the web.[5] In some cases, a test case is included with the task to trigger the bug. This was necessary whenever the method was passed a Socket, TimerTask, ResultSet, or URLConnnection instance.

The code in the method body was most commonly taken directly from one of the questions related to a topic. However, some topics required more creativity because the questions did not include code. For example, the state-test related questions did not contain code which motivated the questioner's need for the state test. Therefore, we created tasks that required knowledge of the state. These tasks each involved writing a method which takes a Timer or URLConnection instance as an argument and uses the instance in a state-specific manner.

2) Example task: To understand better how tasks were constructed, let us look at an example task in more depth. We focus on a task corresponding to the topic "Cannot: Call next on InsertRow." The task involves inserting a new row in a database table via a ResultSet instance and then trying to call the next method.

The ResultSet protocol prohibits scrolling (e.g. calling the next method), while the "cursor is on the insert row." To understand this better, let us look at the state machine diagram show in Figure 2. The cursor position is modeled by the abstract state Position. The Position state has two or-children, CurrentRow and InsertRow, which represent the state of the ResultSet when the cursor is on existing row or on the insert row respectively. Note that the method moveToInsertRow transitions the ResultSet from the CurrentRow state to the InsertRow state. In reverse, the method moveToCurrentRow transitions the object back to the CurrentRow.

A slightly abbreviate version of the code participants were given is shown in Listing 1. Programmers were asked to fix a bug, revealed by a test case, in the insertHarryBovik method. In particular, running the test case results in an SQLException when the next method is called on line 15.

```
1  /**
2   * Precondition: rs is a CONCUR_UPDATABLE ResultSet
3   * to an attached table with at least one row and String
4   * columns labeled, ‘‘first’’ and ‘‘last’’
5   *
6   * Postcondition: Insert a new row with ‘‘Harry’’ in the
7   * ‘‘first’’ column and ‘‘Bovik’’ in the last column. Update
8   * next row’s last name to ‘‘Carnegie’’.
9   */
10 public void insertHarryBovik(ResultSet rs) {
11     rs.moveToInsertRow();
12     rs.updateString("first", "Harry");
13     rs.updateString("last", "Bovik");
14     rs.insertRow();
15     rs.next();
16     rs.updateString("last", "Carnegie");
17     rs.updateRow();
18 }
```

Listing 1. Source code for example task.

To fix the bug participants needed to add just one line in the code. Before calling next, the ResultSet needs to be transitioned to the CurrentRow state by calling the method moveToCurrentRow. As we will see in Section V-B1, this task was surprisingly difficult even for the expert programmers performing the study.

The rest of the tasks have a similar flavor. They require programmers to write new small programs or fix existing small programs involving protocols. All require programmers to navigate the state machine of an underlying object.

3) Study design: We have found that protocols are very challenging for novice programmers or programmers without significant experience using object-oriented libraries and frameworks written in statically typed languages. Therefore we recruited 6 programmers with at least 3 years of professional experience with Java or C#. However, these programmers had never used any of the particular libraries under evaluation. The programmers were recruited via personal contacts.

Participants performed the tasks in a campus laboratory. They worked with a computer that had been prepared with Eclipse and a browser opened to the relevant JavaDoc. Participants were asked to "think aloud." The analysis of this study relies on correctly interpreting what participants were looking for while performing the tasks. Therefore, we followed Ko et al. [18] and asked "what are you looking for?" when participants forgot to think aloud, or their statements were unclear. Participants screens and speech were recorded. The study itself took between 1 and 3 hours, almost all of which was spent performing programming tasks. Task instructions were read to each participant and also provided in written form.

B. Results

In this section, we discuss the results of our observations. These observations address RQ2, "How do programmers approach protocol tasks?" We first describe detailed observations

[5] http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/qualitative-study-tasks.zip

from one particular task and then present the aggregate results from the full study.

1) Example task observations: We introduced the ResultSet insertion task the participants performed in Section V-A2. This task was the most time consuming for the participants — time to completion ranged from 16 minutes to 49 minutes. In addition, the participant observations of this task illustrate well the major results we will discuss in the next section.

Recall from Section V-A2 that participants are debugging a protocol violation. In particular, the next method is called while the ResultSet's cursor is on the insert row. However, none of the participants immediately knew this was the source of the problem.

All participants immediately read and interpreted the error message "invalid cursor state: cannot FETCH NEXT, PRIOR, CURRENT, or RELATIVE, cursor position is unknown." Most participants articulated a rapid-fire set of questions about the details of the error message: e.g. "What is FETCH NEXT?," "Why is the cursor position unknown?" The participants seemed to leave these questions unanswered and focus on the beginning of the error message, "invalid cursor state." The participants recognized that this was protocol related and they asked one of two questions: "What is the cursor status of [ResultSet] rs?" (4 participants) or "Which cursor state does rs need to be in to call next?" (2 participants). As we will discuss later in detail later in this section, these two questions are instances of common question categories.

Regardless of the question asked, all six participants looked first at the method documentation for the next method to see if it could help them answer their question. Unfortunately, the next method documentation does not answer either question. Three participants noted that the documentation states that a SQLException is thrown "if a database access error occurs or this method is called on a closed result set." All three immediately decided neither cited source was the cause of the bug in this case.

The participants' searches diverged from this point forward. Three general categories of searches were used: linear scan of task lines, linear scan of method documentation search, undirected/random search through class documentation.

The fastest strategy, employed by two participants, was to look at the method documentation of each method in the source code one by one. They started at next (line 15) and moved upward to insertRow, then updateString,[6] and finally to moveToInsertRow. These participants looked at the documentation by hovering over the method name inside the Eclipse code editor. This strategy is reasonably natural in an IDE that supports hover documentation, but would require constant switching between editor and webpage documentation if a more traditional editor is used.

The fourth sentence of the ResultSet documentation for moveToInsertRow helps participants identify the state that the result set is in: "Only the updater, getter, and insertRow methods may be called when the cursor is on the insert row." All 3 participants that read this documentation articulated a new understanding of the exception message and articulated a follow up question. One participants said, "Aha! The cursor is on the insert row. How do we get the cursor off the insert row to call next?"

Fortunately for the participants that reached the moveToInsertRow documentation the answer to the follow-up question was immediately evident. To call next, one must call moveToCurrentRow, which both has a parallel name and appears after moveToInsertRow in the documentation.

One participant read the method documentation in the order they appeared on the JavaDoc webpage (the previously discussed participants scanned in the order they appear in the task code), which was the slowest search strategy. This participant looked at the next documentation in the Javadoc generated web page. On the web page, the next method appears first in the Method Detail list. The order of the method documentation matches the order that methods appear in the ResultSet source code. The participant scanned all of the documentation between next and moveToInsertRow which represents 2240 lines of the ResultSet source code and more than 100 methods. Thankfully, much of it is repetitive and could therefore be skimmed. After reaching the moveToInsertRow documentation, this participant acted similarly to the task line searchers.

The remaining three participants, like the method documentation scanner, read the next documentation on the web page. From there these participants skipped around somewhat randomly on the webpage. All three of these participant read at least a few irrelevant sections of method documentation. However, these three eventually found themselves at the top of the webpage at the class level documentation. The penultimate section of this documentation provides a code example that "moves the cursor to the insert row, builds a three-column row, and inserts it into rs and into the data source table using the method insertRow."

After reading the example, the participants compared the example code to the buggy code and noticed the missing call to moveToCurrentRow in the buggy code. The participants read the method documentation for moveToCurrentRow before adding it to insertHarryBovik. One explained he was "trying to figure out if you could call next on the current row?" The observations from this task are illustrative of the aggregate results we discuss next.

2) Aggregate results: To address RQ3 ("What information do programmers seek and have difficulty locating while performing protocol tasks?"), we transcribed the audio recordings, noting the time of every statement made or question asked by the participants. We will refer to anything the participant says as a *quote*. We then watched the video recording and mapped these quotes to blocks of time. Whenever we believed the activity on screen was motivated by a quote, we assigned the block in which it was performed to the quote. This mapping allows me to estimate how much time was spent on each quote.

[6] One participant actually moved down to the updateRow documentation before proceeding upward again to updateString. However, the strategies were otherwise identical.

In the vast majority of cases, the mapping was based on simple temporal ordering — if the activity was performed during or after quote A and before any other quote it was assigned to quote A. In a small number of cases, an activity did not seem to match the preceding quote, and therefore the activity left unassigned. This phenomenon was rare because the experimenter usually noticed when this happened and asked the participant to explain his or her actions. In total, we assigned 87% of participant time to a quote.

We then performed open-coding [27] on the quotes, looking for similar quotes that tended to repeat. Four categories of quotes were particularly common. Each of these categories represents a state search task. In total, 82% of the assigned time (or 71% of the total time) was spent working on the following four categories of search. We list here each general category followed by two specific instances of that category drawn from the transcripts:

A What abstract state is an object in?
- "Is the TimerTask scheduled?"
- "What is the cursor state of [ResultSet] rs?"

B What are the capabilities of an object in state X?
- "Can I schedule a scheduled TimerTask?"
- "What can I do on the insert row?"

C In what state(s) can I do operation Z?
- "When can I call doInput?"
- "Which ResultSets can I update?"

D How do I transition from state X to state Y?
- "How do I get off the insert row to the current row?"
- "Which method schedules the TimerTask?'

These search problems are all specific to protocols, and therefore the protocol tasks are dominated by state search.

To clarify the coding process, consider the two instances of category A listed above. The instance, "Is the TimerTask scheduled," contains the name of an abstract state of TimerTask, "scheduled," so that part of the instance was generalized to "state X." "The TimerTask" refers to a an object so that part of the question was generalized to "an object." Therefore, the question was first coded as "Is the object in state X?" In the second instance, "What is the cursor status of [ResultSet] rs," the "cursor status" refers to the state of the ResultSet. This instance maps directly to "What abstract state is an object in?" The code for the first instance was later merged into this more general category.

Many concrete questions are compositions of several categories. Answering, "What do I need to do to the conn to set doInput?" requires answering general questions C and D. The method doInput can only be set in the disconnected state (C), and the only way to get a disconnected connection is to create a new connection (D). Similarly, answering "What methods can I call on [the object referenced] by [variable] conn?" requires answering a combination of A and B.

We break down the questions and time spent in Figure 3. These charts break down only the 71% subset of time spent on state search activities. As you can see, the only combination categories that appeared in the quotes were A+B and C+D.

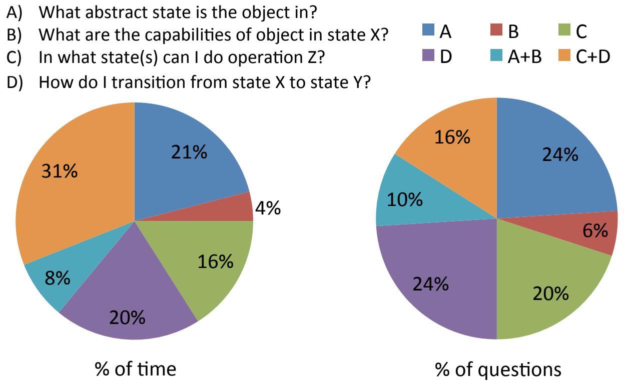


Fig. 3. Question type frequencies.

It's possible to come up with other combinations (e.g. B+D: "I wonder what would happen if I find a transition to state Y?) but harder to envision how they would be useful.

The question types appeared with almost equal frequency, except for category B which was relatively infrequent. We expect category B, which is relatively exploratory, to be more useful in greenfield tasks than the tasks in this study.

A reader who compares the two pie charts will observe that the category C+D questions were relatively time consuming (31% of time was spent on 16% of questions). This relationship held for all 6 participants—C+D questions had the highest average time spent for everyone. When category D questions occur alone, it is possible to guess the method name that will transition the object to the wanted state. To give one trivial but common example, if the state is called "connected" it is likely that you want to call a method called connect. However, when you do not know what state you want to transition to, the implication of the category C component of the question, answering question D requires a global search of the class methods.

Resources. This subsection addresses RQ4, "What resources do programmers use while perform- ing protocol tasks?" Participants were allowed to use any resource they liked. However, participants spent 76% of their total time on documentation webpages or hovering over a method documentation. This result conforms with expectations set by the studies discussed in Section III.

We also noted patterns in the particular documentation looked at by programmers. In 56 out 74 cases (including all 6 programmers in the Result Set insertion example) the programmer looked first to the documentation related to the method call occurring at the exception location to solve their problem (next in the Result example). In 13 of the remaining 18 cases the programmer looked first at the method documentation one line above or below the exception location.The participants never looked at the documentation related to the parameter types, including the receiver type, of the method being called when the exception occurs.

Unfortunately, the exception-location method documentation was not the right place to look for the information developers were seeking. We already discussed the problem with the Result.next documentation, but the ResultSet.get* methods were similarly unhelpful for the "Cannot: read after end" task. Equally commonly, the information needed is buried in the very last element of the documentation, the @throws annotation. This information is not displayed in Eclipse hover documentation by default. It was also often skipped by developers reading the documentation in the web page, even when they were looking for the source of an exception! These findings support tools that push rules necessary for invoking methods to developers, like eMoose directives [7].

Question characteristics. We now return to two of the characteristics discussed in Section IV-D1. Participants performed two tasks that specifically required the participants to determine the state of an unknown instance. In both cases, all participants expressed hope for or requested a state test method. More surprisingly, participants requested state test method in 5 other instances. This further reinforces the advice that state test methods should always be provided.

We mentioned that missing state transitions caused frequent questions. However, type qualifier protocols—in which objects never support certain methods after construction—were very easy for participants. Participants seemed to intuitively understand that a ResultSet is created as scrolling or forward only and cannot be changed thereafter. On the other hand, lifecycle protocols, in which the state transitions only moved in one direction frustrated the participants.

VI. Threats to Validity

We started the forum mining with a large list of classes from the Java Standard Library. These were taken primarily from the results of a single study [2]. Beckman's study used a static analysis to find candidate protocols for manual investigation. This analysis missed protocols whose violations do not result in a thrown exception, nor protocols that check for protocol violations in non-standard ways. The interested reader is referred to Section 2.4 of that paper for further details. More generally, all of the APIs in our study are both libraries and from the "resource programming" domain. The protocol barriers may be different for other types of APIs.

We also do not know exactly how representative the Stack Overflow questions are of actual problems encountered in practice, nor if they really are the most difficult problems. For example, programmers may look to other sources to solve their hardest problems. Similarly, the particular demographic that uses Stack Overflow the most may have different problems than a more representative sample.

The developers who performed the laboratory study were professional engineers, but they were all personal contacts. It is therefore possible that they are very unrepresentative of the population of all skilled developers. Furthermore, the developer sample size was very small. A larger, more representative sample of developers may have needed very different information or very different resources.

Finally, a single experimenter analyzed all of the forum questions, assigning quotes to programmer activity, and categorizing quotes. Another rater would have enabled a reliability assessment and may have caught errors. The question categories may be poorly defined and the quantitative results may be skewed by experimenter biases.

VII. Conclusion

In this study, we identified five common characteristics of the questions about API protocols that developers find particularly problematic. Using the tasks that brought about the problematic questions, we found that experienced developers spent the majority of their time (71%) addressing four types of state searches, some of which are poorly supported by current approaches to documentation.

Our observations suggest that protocol-targeted tools, languages, and verification techniques will be most effective if they enable programmers to efficiently answer the four state search questions. Unfortunately, many of the tools in this area do not directly address any of these questions.

That said, when a protocol is violated some of these tools provide an error message that tells the developer what part of the protocol has been violated. In particular, the messages usually say what abstract state the object is in, thereby answering question A. Unfortunately, we are unaware of any tool that gives the developer this information when there is not an error. This is probably achievable fairly simply for tools that rely on type systems or static analysis, but is much more difficult for dynamic checkers.

The research community has provided substantially less support in answering the other three state search questions (B, C, and D). However, some programming languages support separating members by abstract state which will likely make it easier for developers to answer B and C. Similarly, a first class state change operation in a programming language makes it easier to answer D.

Throughout this paper we discussed many examples in which the information needs of developers do not match the documentation at the location it is needed. In most of the instances the relevant instructions are simply misplaced. We urge writers of documentation to carefully consider how documentation is used when considering its structure. In addition, we believe there is a research opportunity to generate protocol-specific documentation in all of the locations it is needed from simple specifications.

Finally, we mentioned briefly in Section IV-C that answerers sometimes suggested alternative libraries to questioners. These answers were often accepted and/or received many "up-votes" from the Stack Overflow community. This suggests that developers who struggle with protocol violations abandon the APIs. Researchers and practitioners are very interested in what causes tools to be adopted by developers. This study provides evidence that potential adopters can be driven away by difficulty using an API correctly.

VIII. Acknowlegements

This work was supported by NSA lablet contract #H98230-14-C-014, and NSF grant #CCF-1116907.

References

[1] Stephanie Balzer and Thomas R. Gross. Verifying multi-object invariants with relationships. In *ECOOP 2011 – Object-Oriented Programming*, pages 358–382. Springer Berlin Heidelberg, 2011.

[2] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *ECOOP 2011 – Object-Oriented Programming*, pages 2–26. Springer Berlin Heidelberg, 2011.

[3] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 217–226, New York, NY, USA, 2005. ACM.

[4] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 195–219, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, second edition, 2008.

[6] John M Daughtry, Umer Farooq, Jeffrey Stylos, and Brad A Myers. API usability: CHI'2009 special interest group meeting. In *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, pages 2771–2774. ACM, 2009.

[7] Uri Dekel and James D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 320–330, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, ECOOP '04, pages 465–490, London, UK, 2004. Springer-Verlag.

[9] Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive online program analysis. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[13] Benjamin V. Hanrahan, Gregorio Convertino, and Les Nelson. Modeling problem difficulty and expertise in stackoverflow. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work Companion*, CSCW '12, pages 91–94, New York, NY, USA, 2012. ACM.

[14] Ciera Jaspan and Jonatathan Aldrich. Are object protocols burdensome? an empirical study of developer forums. In *Evaluation and Usability of Programming Languages and Tools Workshop (PLATEAU '11)*, 2011.

[15] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *ECOOP 2009 – Object-Oriented Programming*, pages 27–51. Springer Berlin Heidelberg, 2009.

[16] Ciera N.C. Jaspan. *Proper Plugin Protocols*. PhD thesis, Carnegie Mellon University, December 2011. Technical Report: CMU-ISR-11-116.

[17] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM.

[18] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[19] George Kuk. Strategic interaction and knowledge sharing in the kde developer mailing list. *Management Science*, 52(7):1031–1042, 2006.

[20] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 361–370, New York, NY, USA, 2007. ACM.

[21] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest Q&A site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 2857–2866. ACM, 2011.

[22] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3):78–86, 1998.

[23] Chris Parnin and Christoph Treude. Measuring api documentation on the web. In *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, pages 25–30. ACM, 2011.

[24] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16:703–732, 2011.

[25] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013.

[26] J. Sillito, G.C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *Software Engineering, IEEE Transactions on*, 34(4):434–451, 2008.

[27] Anselm L. Strauss. *Qualitative Analysis for Social Scientists*. Cambridge University Press, June 1987.

[28] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, January 1986.

[29] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society.

[30] Jeffrey Stylos and Brad A. Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 105–112, New York, NY, USA, 2008. ACM.

[31] Joshua Sunshine. *Protocol Programmability*. PhD thesis, Carnegie Mellon University, December 2013. CMU-ISR-13-117.

[32] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. Structuring documentation to support state search: A laboratory experiment about protocol programming. In *European Conference on Object Oriented Programming (ECOOP)*, 2014.

[33] Christoph Treude, Ohad Barzilay, and M-A Storey. How do programmers ask and answer questions on the web?: NIER track. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 804–807. IEEE, 2011.

[34] Bogdan Vasilescu, Andrea Capiluppi, and Alexander Serebrenik. Gender, representation and online participation: A quantitative study of stackoverflow. In *International Conference on Social Informatics. ASE*, 2012.

[35] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 218–228, New York, NY, USA, 2002. ACM.

[36] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. Example overflow: Using social media for code recommendation. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 38–42. IEEE, 2012.

Safely Composable Type-Specific Languages

Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, and
Alex Potanin,[1] and Jonathan Aldrich

Carnegie Mellon University and Victoria University of Wellington[1]
{comar, darya, lnistor, bwchung, aldrich}@cs.cmu.edu and alex@ecs.vuw.ac.nz[1]

Abstract. Programming languages often include specialized syntax for common datatypes (e.g. lists) and some also build in support for specific specialized datatypes (e.g. regular expressions), but user-defined types must use general-purpose syntax. Frustration with this causes developers to use strings, rather than structured data, with alarming frequency, leading to correctness, performance, security, and usability issues. Allowing library providers to modularly extend a language with new syntax could help address these issues. Unfortunately, prior mechanisms either limit expressiveness or are not safely composable: individually unambiguous extensions can still cause ambiguities when used together. We introduce *type-specific languages* (TSLs): logic associated with a type that determines how the bodies of *generic literals*, able to contain arbitrary syntax, are parsed and elaborated, hygienically. The TSL for a type is invoked only when a literal appears where a term of that type is expected, guaranteeing non-interference. We give evidence supporting the applicability of this approach and formally specify it with a bidirectionally typed elaboration semantics for the Wyvern programming language.

Keywords: extensible languages; parsing; bidirectional typechecking; hygiene

1 Motivation

Many data types can be seen, semantically, as modes of use of general purpose product and sum types. For example, lists can be seen as recursive sums by observing that a list can either be empty, or be broken down into a product of the *head* element and the *tail*, another list. In an ML-like functional language, sums are exposed as datatypes and products as tuples and records, so list types can be defined as follows:

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

In class-based object-oriented language, objects can be seen as products of their instance data and classes as the cases of a sum type [9]. In low-level languages, like C, structs and unions expose products and sums, respectively.

By defining user-defined types in terms of these general purpose constructs, we immediately benefit from powerful reasoning principles (e.g. induction), language support (e.g. pattern matching) and compiler optimizations. But these semantic benefits often come at a syntactic cost. For example, few would claim that writing a list of numbers as a sequence of `Cons` cells is convenient:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))))
```

Lists are a common data structure, so many languages include *literal syntax* for introducing them, e.g. `[1, 2, 3, 4]`. This syntax is semantically equivalent to the general-purpose syntax shown above, but brings cognitive benefits both when writing and reading code by focusing on the content of the list, rather than the nature of the encoding. Using terminology from Green's cognitive dimensions of notations [8], it is more *terse*, *visible* and *maps more closely* to the intuitive notion of a list. Stoy, in discussing the value of good notation, writes [31]:

> A good notation thus conceals much of the inner workings behind suitable abbreviations, while allowing us to consider it in more detail if we require: matrix and tensor notations provide further good examples of this. It may be summed up in the saying: "A notation is important for what it leaves out."

Although list, number and string literals are nearly ubiquitous features of modern languages, some languages provide specialized literal syntax for other common collections (like maps, sets, vectors and matrices), external data formats (like XML and JSON), query languages (like regular expressions and SQL), markup languages (like HTML and Markdown) and many other types of data. For example, a language with built-in notation for HTML and SQL, supporting type safe *splicing* via curly braces, might define:

```
let webpage : HTML = <html><body><h1>Results for {keyword}</h1>
  <ul id="results">{to_list_items(query(db,
    SELECT title, snippet FROM products WHERE {keyword} in title))}
  </ul></body></html>
```

as shorthand for:

```
let webpage : HTML = HTMLElement(Dict.empty(), [BodyElement(Dict.empty(),
  [H1Element(Dict.empty(), [TextNode("Results for " + keyword)]),
  ULElement((Dict.add Dict.empty() ("id","results")), to_list_items(query(db,
    SelectStmt(["title", "snippet"], "products",
      [WhereClause(InPredicate(StringLit(keyword), "title"))]))))])])
```

When general-purpose notation like this is too cognitively demanding for comfort, but a specialized notation as above is not available, developers turn to run-time mechanisms to make constructing data structures more convenient. Among the most common strategies in these situations, no matter the language paradigm, is to simply use a string representation, parsing it at run-time:

```
1 let webpage : HTML = parse_html("<html><body><h1>Results for "+keyword+"</h1>
2   <ul id=\"results\">" + to_string(to_list_items(query(db, parse_sql(
3     "SELECT title, snippet FROM products WHERE '"+keyword+"' in title")))) +
4   "</ul></body></html>")
```

Though recovering some of the notational convenience of the literal version, it is still more awkward to write, requiring explicit conversions to and from structured representations (`parse_html` and `to_string`, respectively) and escaping when the syntax of the data language interferes with the syntax of string literals (line 2). Such code also causes a number of problems that go beyond cognitive load. Because parsing occurs at run-time, syntax errors will not be discovered statically, causing potential run-time errors in production scenarios. Run-time parsing also incurs performance overhead, particularly relevant when code like this is executed often (as on a heavily-trafficked website). But the most serious issue with this code is that it is highly insecure: it is vulnerable to cross-site scripting attacks (line 1) and SQL injection attacks (line 3). For example, if a user entered the keyword `'; DROP TABLE products --`, the entire product database could be erased. These attack vectors are considered to be two of the most serious security threats on the web today [26]. Although developers are cautioned to sanitize their input, it can be difficult to verify that this was done correctly throughout a codebase. The best way to avoid these problems today is to avoid strings and other similar conveniences and insist on structured representations. Unfortunately, situations like this, where maintaining strong correctness, performance and security guarantees entails significant syntactic overhead, causing developers to turn to less structured solutions that are more convenient, are quite common (as we will discuss in Sec. 5).

Adding new literal syntax into a language is generally considered to be the responsibility of the language's designers. This is largely for technical reasons: not all syntactic forms can unambiguously coexist in the same grammar, so a designer is needed to decide which syntactic forms are available, and what their semantics should be. For example, conventional notations for sets and maps are both delimited by curly braces. When Python introduced set literals, it chose to distinguish them based on whether the literal contained only values (e.g. `{3}`), or key-value pairs (`{"x": 3}`). But this causes an ambiguity with the syntactic form `{ }` – should it mean an empty set or an empty map (called a dictionary in Python)? The designers of Python avoided the ambiguity by choosing the latter interpretation (in this case, for backwards compatibility reasons).

Were this power given to library providers in a decentralized, unconstrained manner, the burden of resolving ambiguities would instead fall on developers who happened to import conflicting extensions. Indeed, this is precisely the situation with SugarJ [6] and other extensible languages generated by Sugar* [7], which allow library providers to extend the base syntax of the host language with new forms in a relatively unconstrained manner. These new forms are imported transitively throughout a program. To resolve syntactic ambiguities that arise, clients must manually augment the composed grammar with new rules that allow them to choose the correct interpretation explicitly. This is both difficult to do, requiring a reasonably thorough understanding of the underlying parser technology (in Sugar*, generalized LR parsing) and increases the cognitive load of using the conflicting notations (e.g. both sets and maps) together because disambiguation tokens must be used. These kinds of conflicts occur in a variety of circumstances: HTML and XML, different variants of SQL, JSON literals and maps, or differing implementations ("desugarings") of the same syntax (e.g. two regular expression engines). Code that uses these common abstractions together is very common in practice [13].

In this work, we will describe an alternative parsing strategy that sidesteps these problems by building into the language only a delimitation strategy, which ensures that ambiguities do not occur. The parsing and elaboration of literal bodies occurs during typechecking, rather than in the initial parsing phase. In particular, the typechecker defers responsibility to library providers, by treating the body of the literal as a term of the *type-specific language (TSL)* associated with the type it is being checked against. The TSL definition is responsible for elaborating this term using only general-purpose syntax. This strategy permits significant semantic flexibility – the meaning of a form like `{ }` can differ depending on its type, so it is safe to use it for empty sets, maps and

JSON literals. This frees these common forms from being tied to the variant of a data structure built into a language's standard library, which may not provide the precise semantics that a programmer needs (for example, Python dictionaries do not preserve key insertion order).

We present our work as a variant of an emerging programming language called Wyvern [22]. To allow us to focus on the essence of our proposal and provide the community with a minimal foundation for future work, the variant of Wyvern we develop here is simpler than the variant we previously described: it is purely functional (there are no effects other than non-termination) and it does not enforce a uniform access principle for objects (fields can be accessed directly), so objects are essentially just recursive labeled products with simple methods. It also adds recursive sum types, which we call *case types*, similar to those found in ML. One can refer to our version of the language as *TSL Wyvern* when the variant being discussed is not clear. Our work substantially extends and makes concrete a mechanism we sketched in a short workshop paper [23].

The paper is organized as a language design for TSL Wyvern:

- In Sec. 2, we introduce TSL Wyvern with a practical example. We introduce both inline and forward referenced literal forms, splicing, case and object types and an example of a TSL definition.
- In Sec. 3, we specify the layout-sensitive concrete syntax of TSL Wyvern with an Adams grammar and introduce the abstract syntax of TSL Wyvern.
- In Sec. 4, we specify the static semantics of TSL Wyvern as a *bidirectionally typed elaboration semantics*, which combines two key technical mechanisms:
    1. **Bidirectional Typechecking**: By distinguishing locations where an expression must synthesize a type from locations where an expression is being analyzed against a known type, we precisely specify where generic literals can appear and how dispatch to a TSL definition (an object with a parse method serving as metadata of a type) occurs.
    2. **Hygienic Elaboration**: Elaboration of literals must not cause the inadvertent capture or shadowing of variables in the context where the literal appears. It must, however, remain possible for the client to do so in those portions of the literal body treated as spliced expressions. The language cannot know *a priori* where these spliced portions will be. We give a clean type-theoretic formulation that achieves of this notion of hygiene.
- In Sec. 5, we gather initial data on how broadly applicable our technique may be by conducting a corpus analysis, finding that existing code often uses strings where specialized syntax might be more appropriate.
- In Sec. 6, we briefly report on the current implementation status of our work.
- We discuss related work in Sec. 7 and conclude in Sec. 8 with a discussion of present limitations and future research directions.

2 Type-Specific Languages in Wyvern

We begin with an example in Fig. 1 showing several different TSLs being used in a fragment of a web application showing search results from a database. We will review this example below to develop intuitions about TSLs in Wyvern; a formal and more detailed description will follow. For clarity of presentation, we color each character by the TSL it is governed by. Black is the base language and comments are in italics.

```
1  let imageBase : URL = <images.example.com>
2  let bgImage : URL = <%imageBase%/background.png>
3  new : SearchServer
4    def resultsFor(searchQuery, page)
5      serve(~) (* serve : HTML -> Unit *)
6        >html
7          >head
8            >title Search Results
9            >style ~
10             body { background-image: url(%bgImage%) }
11             #search { background-color: %darken(‘#aabbcc‘, 10pct)% }
12         >body
13           >h1 Results for <{HTML.Text(searchQuery)}:
14           >div[id="search"]
15             Search again: < SearchBox("Go!")
16           < (* fmt_results : DB * SQLQuery * Nat * Nat -> HTML *)
17              fmt_results(db, ~, 10, page)
18                SELECT * FROM products WHERE {searchQuery} in title
```

Fig. 1: Wyvern Example with Multiple TSLs

```
<literal body here, <inner angle brackets> must be balanced>
{literal body here, {inner braces} must be balanced}
[literal body here, [inner brackets] must be balanced]
‘literal body here, ‘‘inner backticks‘‘ must be doubled‘
’literal body here, ’’inner single quotes’’ must be doubled’
"literal body here, ""inner double quotes"" must be doubled"
12xyz (* no delimiters necessary for number literals; suffix optional *)
```

Fig. 2: Inline Generic Literal Forms

2.1 Inline Literals

Our first TSL appears on the right-hand side of the variable binding on line 1. The variable `imageBase` is annotated with its type, `URL`. This is a named object type declaring several fields representing the components of a URL: its protocol, domain name, port, path and so on (below). We could have created a value of type `URL` using the general-purpose introductory form **`new`**, which *forward references* an indented block of field and method definitions beginning on the line after it appears:

```
objtype URL
  val protocol : String
  val subdomain : String
  (* ... *)
```

```
let imageBase : URL = new
  val protocol = "http"
  val subdomain = "images"
  (* ... *)
```

This is tedious. By associating a TSL with the `URL` type (we will show how later), we can instead introduce precisely this value using conventional notation for URLs by placing it in the *body* of a *generic literal*, `<images.example.com>`. Any other delimited form in Fig. 2 can equivalently be used when the constraints indicated can be obeyed. The type annotation on `imageBase` (or equivalently, ascribed directly to the literal) implies that this literal's *expected type* is `URL`, so the body of the literal (the characters between the angle brackets, in blue) will be governed by the `URL` TSL during the typechecking phase. This TSL will parse the body (at compile-time) and produce an *elaboration*: a Wyvern abstract syntax tree (AST) that explicitly instantiates a new object of type `URL` using general-purpose forms only, as if the above had been written directly.

2.2 Splicing

In addition to supporting conventional notation for URLs, this TSL supports *splicing* another Wyvern expression of type `URL` to form a larger URL. The spliced term is here delimited by percent signs, as seen on line 2 of Fig. 1. The TSL chooses to parse code between percent signs as a Wyvern expression, using its abstract syntax tree (AST) to construct the overall elaboration. A string-based representation of the URL is never constructed at run-time. Note that the delimiters used to go from Wyvern to a TSL are controlled by Wyvern while the TSL controls how to return to Wyvern.

2.3 Layout-Delimited Literals

On line 5 of Fig. 1, we see a call to a function `serve` (not shown) which has type `HTML -> Unit`. Here, `HTML` is a user-defined *case type*, having cases for each HTML tag as well as some other structures, such as text nodes and sequencing. Declarations of some of these cases can be seen on lines 2-6 of Fig. 4 (note that TSL Wyvern also includes simple product types for convenience, written `T1 * T2`). We could again use Wyvern's general-purpose introductory form for case types, e.g. `BodyElement((attrs, child))`. But, as discussed in the introduction, this can be cognitively demanding. Thus, we have associated a TSL with `HTML` that provides a simplified notation for writing HTML, shown being used on lines 6-18 of Fig. 1. This literal body is layout-delimited, rather than delimited by explicit tokens as in Fig. 2, and introduced by a form of *forward reference*, written ~ ("tilde"), on the previous line. Because the forward reference occurs in a position where the expected type is `HTML`, the literal body is governed by that type's TSL. The forward reference will be replaced by the general-purpose term, of type `HTML`, generated by the TSL during typechecking. Because layout was used as a delimiter, there are no syntactic constraints on the body, unlike with inline forms (Fig. 2). For HTML, this is quite useful, as all of the inline forms impose constraints that would cause conflict with some valid HTML, requiring awkward and error-prone escaping. It also avoids issues with leading indentation in multi-line literals, as the parser strips these automatically for layout-delimited literal bodies.

2.4 Implementing a TSL

Portions of the implementation of the TSL for `HTML` are shown on lines 8-15 of Fig. 4. A TSL is associated with a named type using a general mechanism for associating a statically-known value with a named type, called its *metadata*. Type metadata, in this context, is comparable to class annotations in Java or class/type attributes in C#/F# and internalizes the practice of writing metadata using comments, so that it can be checked by the language and accessed programmatically more easily. This can be used for a variety of purposes – to associate documentation with a type, to mark types as being deprecated, and so on. Note that we allow programs to extract the metadata value of a named type `T` programmatically using the form **`metadata`**`[T]`.

For the purposes of this work, metadata values will always be of type `HasTSL`, an object type that declares a single field, `parser`, of type `Parser`. The `Parser` type is an object type declaring a single method, `parse`, that transforms a `ParseStream` extracted from a literal body to a Wyvern AST. An AST is a value of type `Exp`, a case type that encodes the abstract syntax of Wyvern expressions. Fig. 5 shows portions of the decla-

```
casetype HTML
  Empty
  Seq of HTML * HTML
  Text of String
  BodyElement of Attributes * HTML
  StyleElement of Attributes * CSS
  (* ... *)
  metadata = new : HasTSL
    val parser = ~
      start <- '>body'= attributes start>
        fn (attrs, child) => Inj(‘BodyElement‘, Pair(attrs, child))
      start <- '>style'= attributes EXP>
        fn (attrs, e) => ‘StyleElement((%attrs%, %e%))‘
      start <- '<'= EXP>
        fn (e) => ‘%e% : HTML‘
```

Fig. 4: A Wyvern case type with an associated TSL.

```
objtype HasTSL
  val parser : Parser
objtype Parser
  def parse(ps : ParseStream) : Result
  metadata : HasTSL = new
    val parser = (*parser generator*)
casetype Result
  OK of Exp * ParseStream
  Error of String * Location
```

```
casetype Exp
  Var of ID
  Lam of ID * Type * Exp
  Ap of Exp * Exp
  Inj of Id * Exp
  ...
  Spliced of ParseStream
  metadata : HasTSL = new
    val parser = (*quasiquotes*)
```

Fig. 5: Some of the types included in the Wyvern prelude.

rations of these types, which live in the Wyvern *prelude* (a collection of types that are automatically loaded before any other).

Notice, however, that the TSL for `HTML` is not provided as an explicit `parse` method but instead as a declarative grammar. A grammar is specialized notation for defining a parser, so we can implement a grammar-based parser generator as a TSL atop the lower-level interface exposed by `Parser`. We do so using a layout-sensitive grammar formalism developed by Adams [1]. Wyvern is itself layout-sensitive and has a grammar that can be written down using this formalism, as we will discuss, so it is sensible to expose it to TSL providers as well. Most aspects of this formalism are conventional. Each non-terminal (e.g. the designated `start` non-terminal) is defined by a number of disjunctive rules, each introduced using `<-`. Each rule defines a sequence of terminals (e.g. `'>body'`) and non-terminals (e.g. `start`, or one of the built-in non-terminals `ID`, `EXP` or `TYPE`, representing Wyvern identifiers, expressions and types, respectively). Unique to Adams grammars is that each terminal and non-terminal in a rule can also have an optional *layout constraint* associated with it. The layout constraints available are `=` (meaning that the leftmost column of the annotated term must be aligned with that of the parent term), `>` (the leftmost column must be indented further) and `>=` (the leftmost column *may* be indented further). Note that the leftmost column is not simply the first character, in the case of terms that span multiple lines. For example, the production rule of the form $\texttt{A} \rightarrow \texttt{B}^{=}\ \texttt{C}^{\geq}\ \texttt{D}^{>}$ approximately reads as: “Term `B` must be at the same indentation level as term `A`, term `C` may be at the same or a greater indentation level as term `A`, and term `D` must be at an indentation level greater than term `A`’s.” In particular, if `D` contains a `NEWLINE` character, the next line must be indented past the position of the left-most character of A (typically, though not always, constructed so that it must appear at the beginning of a line). There are no constraints relating D to B or C other than the standard sequencing constraint: the first character of D must be further along in the file than the others. Using Adams grammars, the syntax of real-world languages like Python and Haskell can be written declaratively.

Each rule is followed, in an indented block, by a spliced function that generates an elaboration given the elaborations recursively generated by each of the n non-terminals in the rule, ordered left-to-right. Elaborations are of type Exp, which is a case type containing each form in the abstract syntax of Wyvern (as well as an additional case, Spliced, that is used internally), which we will describe later. Here, we show how to generate an elaboration using the general-purpose introductory form for case types (line 11, Inj corresponds to the introductory form for case types) as well as using *quasiquotes* (line 13). Quasiquotes are expressions written in concrete syntax that are not evaluated for their value, but rather evaluate to their corresponding syntax trees. We observe that quasiquotes too fall into the pattern of "specialized notation associated with a type": quasiquotes for expressions, types and identifiers are simply TSLs associated with Exp, Type and ID (Fig. 5). They support the Wyvern concrete syntax as well as an additional delimited form, written with %s, that supports "unquoting": splicing another AST into the one being generated. Again, splicing is safe and structural, not string-based.

We can see how HTML splicing works on lines 12-15: we simply include the Wyvern expression non-terminal EXP in our rule and insert it into our quoted result where appropriate. The type that the spliced Wyvern expression will be expected to have is determined by where it is placed. On line 13 it is known to be CSS by the declaration of HTML, and on line 15, it is known to be HTML by the use of an explicit ascription.

3 Syntax

3.1 Concrete Syntax

We will begin our formal treatment by specifying the concrete syntax of Wyvern declaratively, using the same layout-sensitive formalism that we have introduced for TSL grammars, developed recently by Adams [1]. Adams grammars are useful because they allow us to implement layout-sensitive syntax, like that we've been describing, without relying on context-sensitive lexers or parsers. Most existing layout-sensitive languages (e.g. Python and Haskell) use hand-rolled context-sensitive lexers or parsers (keeping track of, for example, the indentation level using special INDENT and DEDENT tokens), but these are more problematic because they could not be used to generate editor modes, syntax highlighters and other tools automatically. In particular, we will show how the forward references we have described can be correctly encoded without requiring a context-sensitive parser or lexer using this formalism. It is also useful that the TSL for Parser, above, uses the same parser technology as the host language, so that it can be used to generate the quasiquote TSL for Exp more easily.

3.2 Program Structure

The concrete syntax of TSL Wyvern is shown in Fig 6. An example Wyvern program showing several unique syntactic features of TSL Wyvern is shown in Fig. 7 (left).

```
(* programs *)
p → 'objtype'^= ID^> NEWLINE^> objdecls^> metadatadecl^> NEWLINE^> p^=
p → 'casetype'^= ID^> NEWLINE^> casedecls^> metadatadecl^> NEWLINE^> p^=
p → e^=
metadatadecl → ε | 'metadata'^= '='^> e^>
objdecls → ϵ
objdecls → 'val'^= ID^> ':'^> type NEWLINE^> objdecls^>
objdecls → 'def'^= ID^> '('^> typelist^> ')'^> ':'^> type^> NEWLINE^> objdecls^>
casedecls → ϵ
casedecls → ID^= (ϵ | 'of'^> type^>) NEWLINE^> casedecls^>

type → ID^= | type^= '->'^> type^> | type^= '*'^> type^>

e → e̅^=
e → ẽ['~']^= NEWLINE^> chars^>
e → ẽ['new']^= NEWLINE^> m^>
e → ẽ['case(' e̅ ')']^= NEWLINE^> r^>

(* object definitions *)
m → ε
m → 'val'^= ID^> '='^> e^> NEWLINE^> m^=
m → 'def'^= ID^> '('^> idlist^> ')'^> '='^> e^> NEWLINE^> d^=

(* rules for case analysis (case types and products) *)
r → rc | rp
rc → ID^= '('^> ID^> ')'^> '=>'^> e^>
rc → ID^= '('^> ID^> ')'^> '=>'^> e^> NEWLINE^> rc^=
rp → '('^= idlist^> ')'^> '=>'^> e^>

(* expressions containing zero forward references *)
e̅ → ID^=
e̅ → e̅^= ':'^> type^>
e̅ → 'let'^= ID^> (ϵ | ':'^> type^>) '='^> e^> NEWLINE^> e̅^=
e̅ → 'fn'^= '('^> idlist^> ')'^> (ϵ | ':'^> type^>) '=>'^> e̅^>
e̅ → e̅^= '('^> a̅l̅^> ')'^>
e̅ → '('^> a̅l̅^> ')'^>
e̅ → e̅^= '.'^> ID^>
e̅ → 'toast'^= '('^> e̅^> ')'^>
e̅ → 'metadata'^= '['^> ID^> ']'^>
e̅ → inlinelit^=
a̅l̅ → ε | a̅l̅_nonempty^=
a̅l̅_nonempty → e̅^= | e̅^= ','^> a̅l̅_nonempty^>
inlinelit → samedelims^= | matcheddelims^= | numlit^=

(* expressions containing exactly one forward reference *)
ẽ[fwd] → fwd^=
ẽ[fwd] → ẽ[fwd]^= ':'^> type^>
ẽ[fwd] → 'let'^= ID^> (ϵ | ':'^> type^>) '='^> e^> NEWLINE^> ẽ[fwd]^=
ẽ[fwd] → 'let'^= ID^> (ϵ | ':'^> type^>) '='^> ẽ[fwd]^> NEWLINE^> e̅^=
ẽ[fwd] → 'fn'^= idlist^> (ϵ | ':'^> type^>) '=>'^> ẽ[fwd]^>
ẽ[fwd] → ẽ[fwd]^= '('^> a̅l̅^> ')'^>
ẽ[fwd] → e̅^= '('^> a͠l[fwd]^> ')'^>
ẽ[fwd] → '('^> a͠l[fwd]^> ')'^>
ẽ[fwd] → ẽ[fwd]^= '.'^> ID^>
ẽ[fwd] → 'toast'^= '('^> ẽ[fwd]^> ')'^>
a͠l[fwd] → ẽ[fwd]^= | ẽ[fwd]^= ','^> a̅l̅_nonempty^> | e̅^= ','^> a͠l[fwd]^>
```

Fig. 6: Concrete syntax of TSL Wyvern specified as an Adams grammar. Some standard productions and precedence handling rules have been omitted for concision.

```
objtype T
  val y : HTML
let page : HTML->HTML = (fn(x) => ~)
  >html
    >body
      <{x}
page(case(5 : Nat))
  Z(_) => (new : T).y
    val y = ~
      >h1 Zero!
  S(x) => ~
    >h1 Successor!
```

$$
\begin{array}{l}
\mathbf{objtype}[T,(y[\mathbf{named}[HTML]],\emptyset),()];\emptyset;\\
\mathbf{elet}(\mathbf{easc}[\mathbf{arrow}[\mathbf{named}[HTML],\\
\quad \mathbf{named}[HTML]]](\mathbf{elam}(x.\mathbf{lit}[>\mathit{html}\\
\quad >\mathit{body}\\
\qquad <\mathit{\{x\}}])),page.\\
\mathbf{eap}(page;\mathbf{ecase}(\mathbf{easc}[\mathbf{named}[Nat]](\mathbf{lit}[5]))\ \{\\
\quad \mathbf{erule}[Z](\_.\mathbf{eprj}[y](\mathbf{easc}[\mathbf{named}[T](\mathbf{enew}\ \{\\
\qquad \mathbf{eval}[y](\mathbf{lit}[>\mathit{h1\ Zero!}]);\emptyset\}))));\\
\quad \mathbf{erule}[S](x.\mathbf{lit}[>\mathit{h1\ Sucessor!}]);\emptyset\\
\}))
\end{array}
$$

Fig. 7: An example Wyvern program demonstrating all three forward referenced forms. The corresponding abstract syntax is on the right.

The top level of a program (the `p` non-terminal) consists of a series of named type declarations – object types using **`objtype`** or case types using **`casetype`** – followed by an expression, `e`. Each named type declaration can also include a metadata declaration. Metadata is simply an expression associated with the type, used to store TSL logic (and in future work, other metadata). In the grammar, sequences of top-level declarations use the form `p`$^=$ to signify that all the succeeding `p` terms must begin at the same indentation. We do not specify separate compilation here, as this is an orthogonal issue.

3.3 Forward Referenced Blocks

Wyvern makes extensive use of forward referenced blocks to make its syntax clean. In particular, layout-delimited TSLs, **`new`** expressions for introducing objects, and **`case`** expressions for eliminating case types and tuples all make use of forward referenced blocks. Fig. 7 shows these in use (assuming suitable definitions of `Nat` and `HTML`).

Each line in the concrete syntax can contain either zero or one forward references. We distinguish these in the grammar by defining separate non-terminals $\overline{\mathtt{e}}$ and $\widetilde{\mathtt{e}}$`[fwd]`, where the parameter `fwd` is the particular forward reference form that occurs. Note particularly the rule for **`let`** (which permits an expression to span multiple lines and so can be used to support multiple forward references in a single expression).

3.4 Abstract Syntax

The concrete syntax of a Wyvern program, `p`, is parsed to a program in the abstract syntax, ρ, shown in Fig. 8. Forward references are internalized. Note that all literal forms are unified into the abstract literal form **lit**$[\mathit{body}]$, including the layout-delimited form and number literals. The body remains completely unparsed at this stage. The abstract syntax for the example in Fig. 7 is shown to its right and demonstrates the key rewriting done at this stage. Simple product types can be rewritten as object types in this phase. We assume that this occurs so that we can avoid specifying them separately in the remainder of the paper, though we continue to use tuple notation for concision.

4 Bidirectional Typechecking and Elaboration

We will now specify a type system for the abstract syntax in Fig. 8. Conventional type systems are specified using a typing judgement written like $\Gamma \vdash_\Theta e : \tau$, where the typing context, Γ, maps bound variables to types, and the named type context, Θ, maps

$$
\begin{array}{ll}
\rho ::= \theta; e & \tau ::= \textbf{named}[T] \mid \textbf{arrow}[\tau, \tau] \\
\theta ::= \emptyset & \\
\quad \mid\ \textbf{objtype}[T, \omega, e]; \theta & \omega ::= \emptyset \mid \ell[\tau]; \omega \\
\quad \mid\ \textbf{casetype}[T, \chi, e]; \theta & \chi ::= \emptyset \mid C[\tau]; \chi
\end{array}
$$

$$
\begin{array}{lll}
e \ ::= x & \hat{e} \ ::= x & i \ ::= x \\
\quad \mid\ \textbf{easc}[\tau](e) & \quad \mid\ \textbf{hasc}[\tau](\hat{e}) & \quad \mid\ \textbf{iasc}[\tau](i) \\
\quad \mid\ \textbf{elet}(e; x.e) & \quad \mid\ \textbf{hlet}(\hat{e}; x.\hat{e}) & \quad \mid\ \textbf{ilet}(i; x.i) \\
\quad \mid\ \textbf{elam}(x.e) & \quad \mid\ \textbf{hlam}(x.\hat{e}) & \quad \mid\ \textbf{ilam}(x.i) \\
\quad \mid\ \textbf{eap}(e; e) & \quad \mid\ \textbf{hap}(\hat{e}; \hat{e}) & \quad \mid\ \textbf{iap}(i; i) \\
\quad \mid\ \textbf{enew}\,\{m\} & \quad \mid\ \textbf{hnew}\,\{\hat{m}\} & \quad \mid\ \textbf{inew}\,\{\dot{m}\} \\
\quad \mid\ \textbf{eprj}[\ell](e) & \quad \mid\ \textbf{hprj}[\ell](\hat{e}) & \quad \mid\ \textbf{iprj}[\ell](i) \\
\quad \mid\ \textbf{einj}[C](e) & \quad \mid\ \textbf{hinj}[C](\hat{e}) & \quad \mid\ \textbf{iinj}[C](i) \\
\quad \mid\ \textbf{ecase}(e)\,\{r\} & \quad \mid\ \textbf{hcase}(\hat{e})\,\{\hat{r}\} & \quad \mid\ \textbf{icase}(i)\,\{\dot{r}\} \\
\quad \mid\ \textbf{etoast}(e) & \quad \mid\ \textbf{htoast}(\hat{e}) & \quad \mid\ \textbf{itoast}(i) \\
\quad \mid\ \textbf{emetadata}[T] & \quad \mid\ \textbf{hmetadata}[T] & \\
\quad \mid\ \textbf{lit}[\mathit{body}] & \quad \mid\ \textbf{spliced}[e] & \\
m ::= \emptyset & \hat{m} ::= \emptyset & \dot{m} ::= \emptyset \\
\quad \mid\ \textbf{eval}[\ell](e); m & \quad \mid\ \textbf{hval}[\ell](\hat{e}); \hat{m} & \quad \mid\ \textbf{ival}[\ell](i); \dot{m} \\
\quad \mid\ \textbf{edef}[\ell](x.e); m & \quad \mid\ \textbf{hdef}[\ell](x.\hat{e}); \hat{m} & \quad \mid\ \textbf{idef}[\ell](x.i); \dot{m} \\
r \ ::= \emptyset & \hat{r} \ ::= \emptyset & \dot{r} \ ::= \emptyset \\
\quad \mid\ \textbf{erule}[C](x.e); r & \quad \mid\ \textbf{hrule}[C](x.\hat{e}); \hat{r} & \quad \mid\ \textbf{irule}[C](x.i); \dot{r}
\end{array}
$$

Fig. 8: Abstract Syntax of TSL Wyvern programs (ρ), type declarations (θ), types (τ), external terms (e), translational terms ($\hat{e}$) and internal terms (i) and auxiliary forms. Metavariable T ranges over type names, ℓ over object member (field and method) labels, C over case labels, x over variables and *body* over literal bodies. Tuple types are a mode of use of object types, so they are not included in the abstract syntax. For concision, we continue to write unit as $()$ and pairs as (i_1, i_2) in abstract syntax as needed.

type names to their declarations. Such typing judgements do not fully specify whether, when writing a typechecker, the type should be considered an input or an output. In some situations, a type propagates in from the surrounding syntactic context (e.g. when the term appears as a function argument, or an explicit ascription has been provided), so that we simply need to *analyze* e against it. In others, we need to *synthesize* a type for e (e.g. when the term appears at the top-level). Here, this distinction is crucial: a literal can only appear in an analytic context. *Bidirectional type systems* [28] make this distinction explicit by specifying the type system instead using two simultaneously defined typechecking judgements corresponding to these two situations.

To support TSLs, we need to also, simultaneously with this process, perform an elaboration from external terms, which contain literals, to *internal terms*, i, the syntax for which is shown on the right side of Fig. 8. Internal terms contain neither literals nor the form for accessing the metadata of a named type explicitly (the elaboration process inserts the statically known metadata value, tracked by the named type context, directly). This manner of specifying a type-directed mapping from external terms to a smaller collection of internal terms, which are the only terms that are given a dynamic

$$\boxed{\rho \sim \varTheta \rightsquigarrow i : \tau} \qquad \varTheta ::= \emptyset \mid \varTheta, T[\delta, \mu] \quad \delta ::= \ ? \mid \textbf{ot}[\omega] \mid \textbf{ct}[\chi] \qquad \mu ::= \ ? \mid i : \tau$$

$$\frac{\vdash_{\varTheta_0} \theta \sim \varTheta \quad \emptyset \vdash_{\varTheta_0\varTheta} e \rightsquigarrow i \Rightarrow \tau}{\theta; e \sim \varTheta \rightsquigarrow i : \tau}\ \textit{Compile}$$

$$\boxed{\vdash_{\varTheta} \theta \sim \varTheta}$$

$$\frac{T \notin \text{dom}(\varTheta) \quad \vdash_{\varTheta, T[?,?]} \omega \quad \emptyset \vdash_{\varTheta, T[\textbf{ot}[\omega],?]} e_m \rightsquigarrow i_m \Rightarrow \tau_m \quad \vdash_{\varTheta, T[\textbf{ot}[\omega], i_m : \tau_m]} \theta \rightsquigarrow \varTheta'}{\vdash_{\varTheta} \textbf{objtype}[T, \omega, e_m]; \theta \sim T[\textbf{ot}[\omega], i_m : \tau_m]; \varTheta'}\ \textit{OT}$$

$$\frac{T \notin \text{dom}(\varTheta) \quad \vdash_{\varTheta, T[?,?]} \chi \quad \emptyset \vdash_{\varTheta, T[\textbf{ct}[\chi],?]} e_m \rightsquigarrow i_m \Rightarrow \tau_m \quad \vdash_{\varTheta, T[\textbf{ct}[\chi], i_m : \tau_m]} \theta \rightsquigarrow \varTheta'}{\vdash_{\varTheta} \textbf{casetype}[T, \chi, e_m]; \theta \sim T[\textbf{ct}[\chi], i_m : \tau_m]; \varTheta'}\ \textit{CT}$$

$$\boxed{\vdash_{\varTheta} \omega} \quad \frac{\ell \notin \text{dom}(\omega) \quad \vdash_{\varTheta} \tau \quad \vdash_{\varTheta} \omega}{\vdash_{\varTheta} \ell[\tau]; \omega}\ \textit{M-decl} \qquad \boxed{\vdash_{\varTheta} \chi} \quad \frac{C \notin \text{dom}(\chi) \quad \vdash_{\varTheta} \tau \quad \vdash_{\varTheta} \chi}{\vdash_{\varTheta} C[\tau]; \chi}\ \textit{C-decl}$$

$$\boxed{\vdash_{\varTheta} \tau} \quad \frac{T[\delta, \mu] \in \varTheta}{\vdash_{\varTheta} \textbf{named}[T]}\ \textit{Ty-named} \qquad \frac{\vdash_{\varTheta} \tau_1 \quad \vdash_{\varTheta} \tau_2}{\vdash_{\varTheta} \textbf{arrow}[\tau_1, \tau_2]}\ \textit{Ty-arrow}$$

Fig. 9: Typechecking and elaboration of programs, ρ. Note that type declarations can only be recursive, not mutually recursive, with these rules. The prelude $\varTheta_0$ (see Fig. 5) defines mutually recursive types, so we cannot write a θ_0 corresponding to $\varTheta_0$ given the rules above. For concision, the rules to support mutual recursion as well as omitted rules for empty declarations are available in a technical report [24].

semantics, is related to the Harper-Stone elaboration semantics for Standard ML [10]. Note that both terms share a type system.

Our static semantics are thus formulated by combining these two ideas, forming a *bidirectionally typed elaboration semantics*. The judgement $\varGamma \vdash_{\varTheta} e \rightsquigarrow i \Rightarrow \tau$ means that under typing context $\varGamma$ and named type context $\varTheta$, external term e elaborates to internal term i and synthesizes type τ. The judgement $\varGamma \vdash_{\varTheta} e \rightsquigarrow i \Leftarrow \tau$ is analagous but for situations where we are analyzing e against type τ.

4.1 Programs and Type Declarations

Before considering these judgements in detail, let us briefly discuss the steps leading up to typechecking and elaboration of the top-level term, specified by the compilation judgement, $\rho \sim \varTheta \rightsquigarrow i : \tau$, defined in Fig. 9. We first load the prelude, $\varTheta_0$ (see Fig. 5), then validate the provided user-defined type declarations, θ, to produce a corresponding named typed context, $\varTheta$. During this process, we synthesize a type for the associated metadata terms (under the empty typing context) and store their elaborations in the type context $\varTheta$ (we do not evaluate the elaboration to a value immediately here, though in a language with effects, the choice of when to evaluate the term is important). Note that type names must be unique (we plan to use a URI-based mechanism in practice). Finally, the top-level external term must synthesize a type τ and produce an elaboration i under an empty typing context and a named type context combining the prelude with the named type context induced by the user-defined types, written $\varTheta_0\varTheta$.

$$\boxed{\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau} \quad \boxed{\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau} \qquad \Gamma ::= \emptyset \mid \Gamma, x : \tau$$

$$\frac{\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau}{\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau}\;\textit{T-syn-to-ana} \qquad \frac{\vdash_\Theta \tau \quad \Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau}{\Gamma \vdash_\Theta \mathbf{easc}[\tau](e) \rightsquigarrow \mathbf{iasc}[\tau](i) \Rightarrow \tau}\;\textit{T-asc}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_\Theta x \rightsquigarrow x \Rightarrow \tau}\;\textit{T-var} \qquad \frac{\Gamma \vdash_\Theta e_1 \rightsquigarrow i_1 \Rightarrow \tau_1 \quad \Gamma, x : \tau_1 \vdash_\Theta e_2 \rightsquigarrow i_2 \Rightarrow \tau}{\Gamma \vdash_\Theta \mathbf{elet}(e_1; x.e_2) \rightsquigarrow \mathbf{ilet}(i_1; x.i_2) \Rightarrow \tau}\;\textit{T-let}$$

$$\frac{\Gamma, x : \tau_1 \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau_2}{\Gamma \vdash_\Theta \mathbf{elam}(x.e) \rightsquigarrow \mathbf{ilam}(x.i) \Leftarrow \mathbf{arrow}[\tau_1, \tau_2]}\;\textit{T-abs}$$

$$\frac{\Gamma \vdash_\Theta e_1 \rightsquigarrow i_1 \Rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_\Theta e_2 \rightsquigarrow i_2 \Leftarrow \tau_1}{\Gamma \vdash_\Theta \mathbf{eap}(e_1; e_2) \rightsquigarrow \mathbf{iap}(i_1; i_2) \Rightarrow \tau_2}\;\textit{T-ap}$$

$$\frac{T \neq ParseStream \quad T[\mathbf{ot}[\omega], \mu] \in \Theta \quad \Gamma \vdash^T_\Theta m \rightsquigarrow \dot{m} \Leftarrow \omega}{\Gamma \vdash_\Theta \mathbf{enew}\,\{m\} \rightsquigarrow \mathbf{inew}\,\{\dot{m}\} \Leftarrow \mathbf{named}[T]}\;\textit{T-new}$$

$$\frac{\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \mathbf{named}[T] \quad T[\mathbf{ot}[\omega], \mu] \in \Theta \quad \ell[\tau] \in \omega}{\Gamma \vdash_\Theta \mathbf{eprj}[\ell](e) \rightsquigarrow \mathbf{iprj}[\ell](i) \Rightarrow \tau}\;\textit{T-prj}$$

$$\frac{T[\mathbf{ct}[\chi], \mu] \in \Theta \quad C[\tau] \in \chi \quad \Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau}{\Gamma \vdash_\Theta \mathbf{einj}[C](e) \rightsquigarrow \mathbf{iinj}[C](i) \Leftarrow \mathbf{named}[T]}\;\textit{T-inj}$$

$$\frac{\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \mathbf{named}[T] \quad T[\mathbf{ct}[\chi], \mu] \in \Theta \quad \Gamma \vdash_\Theta r \rightsquigarrow \dot{r} \Leftarrow \chi \Rightarrow \tau}{\Gamma \vdash_\Theta \mathbf{ecase}(e)\,\{r\} \rightsquigarrow \mathbf{icase}(i)\,\{\dot{r}\} \Rightarrow \tau}\;\textit{T-case}$$

$$\frac{\Theta_0 \subset \Theta \quad \Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau}{\Gamma \vdash_\Theta \mathbf{etoast}(e) \rightsquigarrow \mathbf{itoast}(i) \Rightarrow \mathbf{named}[Exp]}\;\textit{T-toast}$$

$$\frac{T[\delta, i : \tau] \in \Theta}{\Gamma \vdash_\Theta \mathbf{emetadata}[T] \rightsquigarrow i \Rightarrow \tau}\;\textit{T-metadata}$$

$$\boxed{\Gamma \vdash^T_\Theta m \rightsquigarrow \dot{m} \Leftarrow \omega} \qquad \frac{}{\Gamma \vdash^T_\Theta \emptyset \rightsquigarrow \emptyset \Leftarrow \emptyset}\;\textit{T-unit}$$

$$\frac{\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau \quad \Gamma \vdash^T_\Theta m \rightsquigarrow \dot{m} \Leftarrow \omega}{\Gamma \vdash^T_\Theta \mathbf{eval}[\ell](e); m \rightsquigarrow \mathbf{ival}[\ell](i); \dot{m} \Leftarrow \ell[\tau]; \omega}\;\textit{T-val}$$

$$\frac{\Gamma, x : \mathbf{named}[T] \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau \quad \Gamma \vdash^T_\Theta m \rightsquigarrow \dot{m} \Leftarrow \omega}{\Gamma \vdash^T_\Theta \mathbf{edef}[\ell](x.e); m \rightsquigarrow \mathbf{idef}[\ell](x.i); \dot{m} \Leftarrow \ell[\tau]; \omega}\;\textit{T-def}$$

$$\boxed{\Gamma \vdash_\Theta r \rightsquigarrow \dot{r} \Leftarrow \chi \Rightarrow \tau} \qquad \frac{}{\Gamma \vdash_\Theta \emptyset \rightsquigarrow \emptyset \Leftarrow \emptyset \Rightarrow \tau}\;\textit{T-void}$$

$$\frac{\Gamma, x : \tau_1 \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau_2 \quad \Gamma \vdash_\Theta r \rightsquigarrow \dot{r} \Leftarrow \chi \Rightarrow \tau_2}{\Gamma \vdash_\Theta \mathbf{erule}[C](x.e); r \rightsquigarrow \mathbf{irule}[C](x.i); \dot{r} \Leftarrow C[\tau_1]; \chi \Rightarrow \tau_2}\;\textit{T-rule}$$

Fig. 10: Statics for external terms, e. The rule for literals is shown in Fig. 11.

4.2 External Terms

The bidirectional typechecking and elaboration rules for external terms are specified beginning in Fig. 10. Most of the rules are standard for a simply typed lambda calculus with labeled sums and labeled products, and the elaborations are direct to a corresponding internal form. We refer the reader to standard texts on type systems (e.g. [9]) to understand the basic constructs, and to course material[1] on bidirectional typechecking for background. In our presentation, as in many simple formulations, all introductory forms are analytic and all elimination forms are synthetic, though this can be relaxed in practice to support some additional idioms.

The introductory form for object types, **enew** $\{m\}$, prevents the manual introduction of parse streams (only the semantics can introduce parse streams, to permit us to enforce hygiene, as we will discuss below). The auxiliary judgement $\Gamma \vdash^{T}_{\Theta} m \rightsquigarrow \dot{m} \Leftarrow \omega$ analyzes the member definitions m against the member declarations ω while rewriting them to the internal member definitions, $\dot{m}$. Method definitions involve a self-reference, so the judgement keeps track of the type name, T. We implicitly assume that member definitions and declarations are congruent up to reordering.

The introduction form for case types is written **einj**$[C](e)$, where C is the case name and e is the associated data. The type of the data associated with each case is stored in the case type's declaration, χ. Because the introductory form is analytic, multiple case types can use the same case names (unlike in, for example, ML). The elimination form, **ecase**(e) $\{r\}$, performs simple exhaustive case analysis (we leave support for nested pattern matching as future work) using the auxiliary judgement $\Gamma \vdash_{\Theta} r \rightsquigarrow \dot{r} \Leftarrow \chi \Rightarrow \tau$, which checks that each case in χ appears in a rule in the rule sequence r, elaborating it to the internal rule sequence $\dot{r}$. Every rule must synthesize the same type, τ.

The rule *T-metadata* shows how the appropriate metadata is extracted from the named type context and inserted directly in the elaboration. We will return to the rule *T-toast* when discussing hygiene.

4.3 Literals

In the example in Fig. 4, we showed a TSL being defined using a parser generator based an Adams grammars. As we noted, a parser generator can itself be seen as a TSL for a parser, and a parser is the fundamental construct that becomes associated with a type to form a TSL. The declaration for the prelude type `Parser`, shown in Fig. 5, shows that it is an object type with a `parse` function taking in a `ParseStream` and producing a `Result`, which is a case type that indicates either that parsing succeeded, in which case an elaboration of type `Exp` is paired with the remaining parse stream (to allow one parser to call another), or that parsing failed, in which case an error message and location is provided. This function is called by the typechecker when analyzing the literal form, as specified by the key rule of our system, *T-lit*, shown in Fig. 11. Note that we do not explicitly handle failure in the specification, but in practice we would use the data provided in the failure case to report the error to the user.

The rule *T-lit* operates as follows:

1. This rule requires that the prelude is available. For technical reasons, we include a check that the prelude was actually included in the named type context.

[1] `http://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf`

$$\frac{\begin{array}{c}\Theta_0 \subset \Theta \quad T[\delta, i_m : \mathit{HasTSL}] \in \Theta \quad \texttt{parsestream}(\mathit{body}) = i_{ps} \\ \textbf{iap}(\textbf{iprj}[\mathit{parse}](\textbf{iprj}[\mathit{parser}](i_m)); i_{ps}) \Downarrow \textbf{iinj}[\mathit{OK}]((i_{ast}, i'_{ps})) \\ i_{ast} \uparrow \hat{e} \quad \Gamma; \emptyset \vdash_\Theta \hat{e} \rightsquigarrow i \Leftarrow \textbf{named}[T]\end{array}}{\Gamma \vdash_\Theta \textbf{lit}[\mathit{body}] \rightsquigarrow i \Leftarrow \textbf{named}[T]} \; \textit{T-lit}$$

Fig. 11: Statics for external terms, e, continued. This is the key rule (described below).

2. The metadata of the type the literal is being checked against, which must be of type *HasTSL*, is extracted from the named type context. Note that in a language with subtyping or richer forms of type equality, which would be necessary for situations where the metadata might serve other roles, the check that i_m defines a TSL would perform this check explicitly (as an additional premise).
3. A parse stream, i_{ps}, which is an internal term of type **named**$[ParseStream]$, is generated from the body of the literal. This is an object that allows the TSL to read the body and supports some additional conveniences, discussed further below.
4. The `parse` method is called with this parse stream. If it produces the appropriate case containing a *reified elaboration*, i_{ast} (of type `Exp`) and the remaining parse stream, i'_{ps}, then parsing was successful. Note that we use shorthand for pairs in the rule for concision, and the relation $i \Downarrow i'$ defines evaluation to a value (the maximal transitive closure, if it exists, of the small-step evaluation relation in Fig. 15).
5. The reified elaboration is *dereified* into a corresponding *translational term*, $\hat{e}$, as specified in Fig. 12. The syntax for translational terms mirrors that of external terms, but does not include literal forms. It adds the form **spliced**$[e]$, representing an external term spliced into a literal body.

   The key rule is *U-Spl*. The only way to generate a translational term of this form is by asking for (a portion of) a parse stream to be parsed as a Wyvern expression. The reified form, unlike the translational form it corresponds to, does not contain the expression itself, but rather just the portion of the parse stream that should be treated as spliced. Because parse streams (and thus portions thereof) can originate only metatheoretically (i.e. from the compiler), we know that e must be an external term written concretely by the TSL client in the body of the literal being analyzed. This is key to guaranteeing hygiene in the final step, below.

   The convenience methods `parse_exp` and `parse_id` return a value having this reified form corresponding to the first external term found in the parse stream (but, as just described, not necessarily the term itself) paired with the remainder of the parse stream. These methods themselves are not treated specially by the compiler but, for convenience, are associated with `ParseStream`.
6. The final step is to typecheck and elaborate this translational term. This involves the bidirectional typing judgements shown in Fig. 14. This judgement has a form similar to that for external terms, but with the addition of an "outer typing context", written Γ_{out} in the rules. This holds the context that the literal appeared in, so that the "main" typing context can be emptied to ensure that elaborations is hygienic, as we will describe next. Each rule in Fig. 10 should be thought of as having a corresponding rule in Fig. 14. Two examples are shown for concision.

$$\boxed{i \uparrow \hat{e}} \qquad \frac{i_{id} \uparrow x}{\textbf{iinj}[Var](i_{id}) \uparrow x}\ \textit{U-Var}$$

$$\frac{i_1 \uparrow \tau \quad i_2 \uparrow \hat{e}}{\textbf{iinj}[Asc]((i_1, i_2)) \uparrow \textbf{hasc}[\tau](\hat{e})}\ \textit{U-Asc}$$

$$\frac{i_{id} \uparrow x \quad i \uparrow \hat{e}}{\textbf{iinj}[Lam]((i_{id}, i)) \uparrow \textbf{hlam}(x.\hat{e})}\ \textit{U-Lam}$$

$$\frac{i_1 \uparrow \hat{e}_1 \quad i_2 \uparrow \hat{e}_2}{\textbf{iinj}[Ap]((i_1, i_2)) \uparrow \textbf{hap}(\hat{e}_1, \hat{e}_2)}\ \textit{U-Ap}$$

$$\ldots$$

$$\frac{\texttt{body}(i_{ps})\text{=}\textit{body} \quad \texttt{eparse}(\textit{body})\text{=}e}{\textbf{iinj}[Spliced](i_{ps}) \uparrow \textbf{spliced}[e]}\ \textit{U-Spl}$$

$$\boxed{i \uparrow \tau} \qquad \frac{i_{id} \uparrow T}{\textbf{iinj}[Named](i_{id}) \uparrow \textbf{named}[T]}\ \textit{U-N}$$

$$\frac{i_1 \uparrow \tau_1 \quad i_2 \uparrow \tau_2}{\textbf{iinj}[Arrow]((i_1, i_2)) \uparrow \textbf{arrow}[\tau_1, \tau_2]}\ \textit{U-A}$$

Fig. 12: Dereification rules, used by rule *T-lit* (above) to determine the translational term encoded by the internal term of type **named**$[Exp]$. We assume a bijection between internal terms of type **named**$[ID]$ (written i_{id}) and variables, type names and case and member labels.

$$\boxed{i \downarrow i} \qquad \frac{x \downarrow i_{id}}{x \downarrow \textbf{iinj}[Var](i_{id})}\ \textit{R-Var}$$

$$\frac{\tau \downarrow i_1 \quad i \downarrow i_2}{\textbf{iasc}[\tau](i) \downarrow \textbf{iinj}[Asc]((i_1, i_2))}\ \textit{R-Asc}$$

$$\frac{x \downarrow i_{id} \quad i \downarrow i'}{\textbf{ilam}(x.i) \downarrow \textbf{iinj}[Lam]((i_{id}, i'))}\ \textit{R-Lam}$$

$$\frac{i_1 \downarrow i'_1 \quad i_2 \downarrow i'_2}{\textbf{iap}(i_1; i_2) \downarrow \textbf{iinj}[Ap]((i'_1, i_2))}\ \textit{R-Ap}$$

$$\ldots$$

$$\boxed{\tau \downarrow i} \qquad \frac{T \downarrow i_{id}}{\textbf{named}[T] \downarrow \textbf{iinj}[Named](i_{id})}\ \textit{R-N}$$

$$\frac{\tau_1 \downarrow i_1 \quad \tau_2 \downarrow i_2}{\textbf{arrow}[\tau_1, \tau_2] \downarrow \textbf{iinj}[Arrow]((i_1, i_2))}\ \textit{R-A}$$

Fig. 13: Reification rules, used by the **itoast** ("to AST") operator (Fig. 15) to permit generating an internal term of type **named**$[Exp]$ corresponding to the value of the argument (a form of serialization).

4.4 Hygiene

A concern with any term rewriting system is *hygiene* – how should variables in the elaboration be bound? In particular, if the rewriting system generates an *open term*, then it is making assumptions about the names of variables in scope at the site where the TSL is being used, which is incorrect. Those variables should only be identifiable up to alpha renaming. Only the *user* of a TSL knows which variables are in scope. The strictest rule would simply reject all open terms, but this would then, given our setting, prevent even spliced terms from referring to local variables. These are written by the TSL client, who is aware of variable bindings at the use site, so this should be permitted.

Furthermore, the variables in spliced terms should be bound as the client expects. The elaboration should not be able to surreptitiously or accidentally shadow variables in spliced terms that may be otherwise bound at the use site (e.g. by introducing a variable `tmp` outside a spliced term that "leaks" into the spliced term).

The solution to both of these issues, given what we have outlined above, is now quite simple: we have constructed the system so that we know which sub-terms originate from the TSL client, marking them as **spliced**$[e]$. These terms are permitted to refer only to

$$\boxed{\Gamma; \Gamma \vdash_{\Theta} \hat{e} \rightsquigarrow i \Rightarrow \tau} \quad \boxed{\Gamma; \Gamma \vdash_{\Theta} \hat{e} \rightsquigarrow i \Leftarrow \tau}$$

$$\frac{x : \tau \in \Gamma}{\Gamma_{\text{out}}; \Gamma \vdash_{\Theta} x \rightsquigarrow x \Rightarrow \tau}\ \textit{H-var} \qquad \frac{\Gamma_{\text{out}}; \Gamma, x : \tau_1 \vdash_{\Theta} \hat{e} \rightsquigarrow i \Leftarrow \tau_2}{\Gamma_{\text{out}}; \Gamma \vdash_{\Theta} \mathbf{hlam}(x.\hat{e}) \rightsquigarrow \mathbf{ilam}(x.i) \Leftarrow \mathbf{arrow}[\tau_1, \tau_2]}\ \textit{H-abs}$$

$$\cdots$$

$$\frac{\Gamma_{\text{out}} \vdash_{\Theta} e \rightsquigarrow i \Leftarrow \tau}{\Gamma_{\text{out}}; \Gamma \vdash_{\Theta} \mathbf{spliced}[e] \rightsquigarrow i \Leftarrow \tau}\ \textit{H-spl-A} \qquad \frac{\Gamma_{\text{out}} \vdash_{\Theta} e \rightsquigarrow i \Rightarrow \tau}{\Gamma_{\text{out}}; \Gamma \vdash_{\Theta} \mathbf{spliced}[e] \rightsquigarrow i \Rightarrow \tau}\ \textit{H-spl-S}$$

Fig. 14: Statics for translational terms, $\hat{e}$. Each rule in Fig. 10 corresponds to an analagous rule here by threading the outer context through opaquely (e.g. the rules for variables and functions, shown here). The outer context is only used by the rules for **spliced**$[e]$, representing external terms that were spliced into TSL bodies. Note that elaboration is implicitly capture-avoiding here (see Sec. 6).

$$\boxed{i \mapsto i} \qquad \cdots \qquad \frac{i \mapsto i'}{\mathbf{itoast}(i) \mapsto \mathbf{itoast}(i')}\ \textit{D-Toast-1} \qquad \frac{i\ \texttt{val} \quad i \downarrow i'}{\mathbf{itoast}(i) \mapsto i'}\ \textit{D-Toast-2}$$

Fig. 15: Dynamics for internal terms, i. Only internal terms have a dynamic semantics. Most constructs in TSL Wyvern are standard and omitted, as our focus in this paper is on the statics. The only novel internal form, **itoast**(i), extracts an AST (of type **named**$[Exp]$) from the value of i, shown.

variables in the client's context, Γ_{out}, as seen in the premises of the two rules pertaining to this form (one for analysis, one for synthesis). The portions of the elaboration that aren't marked in this way were generated by the TSL provider, so they can refer only to variables introduced earlier in the elaboration, tracked by the context Γ, initially empty. The two are kept separate. If the TSL wishes to introduce values into spliced terms, it must do so by via a function application (as in the TSL for `Parser` discussed earlier), ensuring that the client has full control over variable binding.

4.5 From Values to ASTs

By this formulation, elaborations containing free variables are always erroneous. In some rewriting systems, a free variable is not an error, but are instead replaced with the AST corresponding to the value of the variable at the generation site. We permit this explicitly by including the form `toast(e)`. This simply takes the value of `e` and reifies it, producing a term of type `Exp`, as specified in Figs. 15 and Fig. 13. The rules for reification, used here, and dereification, used in the literal rule above, are dual.

The TSL associated with `Exp`, implementing quasiquotes, can perform free variable analysis and insert this form automatically, so they need not be inserted manually in most cases. That is, `Var('x') : Exp` elaborates to x which is ill-typed in an empty context, `'x' : Exp` produces the translational term **htoast**(**spliced**$[x]$), which will elaborate to **itoast**(x) in the context where the quotation appears (i.e. in the TSL definition), thus behaving as described without requiring that quotations are entirely implemented by the language. This can be seen as a form of serialization and could be implemented as a library using reflection or compile-time metaprogramming techniques (e.g. [20]).

$$\boxed{\Gamma \vdash_{\Theta} i \Rightarrow \tau} \quad \boxed{\Gamma \vdash_{\Theta} i \Leftarrow \tau} \qquad \cdots \qquad \frac{T[\mathbf{ot}[\omega], \mu] \in \Theta \quad \Gamma \vdash_{\Theta}^{T} \dot{m} \Leftarrow \omega}{\Gamma \vdash_{\Theta} \mathbf{inew}\,\{\dot{m}\} \Leftarrow \mathbf{named}[T]}\ \textit{IT-new}$$

Fig. 16: Statics for internal terms, i. Each rule in Fig. 10 except *T-metadata* corresponds to an analogous rule here by removing the elaboration portion. Only the rule for object introduction differs, in that we no longer restrict the introduction of parse streams (internal terms are never written directly by users of the language).

4.6 Metatheory

The semantics we have defined constitute a type safe language. We will outline the key theorems and lemmas here, referring the reader to an accompanying technical report for fuller details [24]. The two key theorems are: internal type safety, and type preservation of the elaboration process.

To prove internal type safety, we must define a bidirectional typing judgement for the internal language, shown and described in Fig. 16 (by the external type preservation theorem, we should never need to explicitly implement this, however). We must also define a well-formedness judgement for named type contexts (not shown).

Theorem 1 (Internal Type Safety). *If* $\vdash \Theta$ *and* $\emptyset \vdash_{\Theta} i \Leftarrow \tau$ *or* $\emptyset \vdash_{\Theta} i \Rightarrow \tau$*, then either* i `val` *or* $i \mapsto i'$ *such that* $\emptyset \vdash_{\Theta} i' \Leftarrow \tau$.

Proof. The dynamics, which we omit for concision, are standard, so the proof is by a standard preservation and progress argument. The only interesting case of the proof involves **etoast**(e), for which we need the following lemma.

Lemma 1 (Reification). *If* $\Theta_0 \subset \Theta$ *and* $\emptyset \vdash_{\Theta} i \Leftarrow \tau$ *then* $i \downarrow i'$ *and* $\emptyset \vdash_{\Theta} i' \Leftarrow$ ***named***$[Exp]$.

Proof. The proof is by a straightforward induction. Analagous lemmas about reification of identifiers and types are similarly straightforward. □

If the elaboration of a closed, well-typed external term generates an internal term of the same type, then internal type safety implies that evaluation will not go wrong, achieving type safety. We generalize this argument to open terms by defining a well-formedness judgement for contexts (not shown). The relevant theorem is below:

Theorem 2 (External Type Preservation). *If* $\vdash \Theta$ *and* $\vdash_{\Theta} \Gamma$ *and* $\Gamma \vdash_{\Theta} e \rightsquigarrow i \Leftarrow \tau$ *or* $\Gamma \vdash_{\Theta} e \rightsquigarrow i \Rightarrow \tau$ *then* $\Gamma \vdash_{\Theta} i \Leftarrow \tau$.

Proof. We proceed by inducting over the the typing derivation. Nearly all the elaborations are direct, so the proof is by straightforward applications of induction hypotheses and lemmas about well-formed contexts. The only cases of note are:

- $e =$ **enew** $\{m\}$. Here the corresponding rule for the elaboration is identical but more permissive, so the induction hypothesis applies.
- $e =$ **emetadata**$[T]$. Here, the elaboration generates the metadata value directly. Well-formedness of Θ implies that the metadata term is of the type assigned.

- $e = \textbf{lit}[body]$. Here, we need to apply internal type safety as well as a mutually defined type preservation lemma about translational terms, below.

Lemma 2 (Translational Type Preservation). *If $\vdash \Theta$ and $\vdash_\Theta \Gamma_{out}$ and $\vdash_\Theta \Gamma$ and $dom(\Gamma_{out}) \cap dom(\Gamma) = \emptyset$ (which we can assume implicitly due to alpha renaming) and $\Gamma_{out}; \Gamma \vdash_\Theta \hat{e} \rightsquigarrow i \Leftarrow \tau$ or $\Gamma_{out}; \Gamma \vdash_\Theta \hat{e} \rightsquigarrow i \Rightarrow \tau$ then $\Gamma_{out}\Gamma \vdash_\Theta i \Leftarrow \tau$.*

Proof. The proof by induction over the typing derivation follows the same outline as above for all the shared cases. The outer context is threaded through opaquely when applying the inductive hypothesis. The only rules of note are the two for the spliced external terms, which require applying the external type preservation theorem recursively. This is well-founded by a metric measuring the size of the spliced external term, written in concrete syntax, since we know it was derived from a portion of the literal body. □

Moving up to the level of programs, we can prove the correctness of compilation theorem below. Together, this implies that derivation of the compilation judgement produces an internal term that does not go wrong.

Theorem 3 (Compilation). *If $\rho \sim \Theta \rightsquigarrow i : \tau$ then $\vdash \Theta$ and $\emptyset \vdash_\Theta i \Leftarrow \tau$.*

Proof. We simply need a lemma about checking type declarations and the result follows straightforwardly.

Lemma 3 (Type Declaration). *If $\vdash_{\Theta_0} \theta \sim \Theta$ then $\vdash \Theta_0\Theta$.*

Proof. The proof is a simple induction using the definition of $\vdash \Theta$ (not shown).

4.7 Decidability

Because we are executing user-defined parsers during typechecking, we do not have a straightforward statement of decidability (i.e. termination) of typechecking: the parser might not terminate, because TSL Wyvern is not a total language (due to self-reference in methods). Indecidability of typechecking is strictly for this reason. Typechecking of terms not containing literals is guaranteed to terminate. Termination of parsers and parser generators has previously been studied (e.g. [15]) and the techniques can be applied to user-defined parsing code to increase confidence in termination. Few compilers, even those with high demands for correctness (e.g. CompCert [17]), have made it a priority to fully verify and prove termination of the parser, because it is perceived that most bugs in compilers arise due to incorrect optimization passes, not initial parsing.

5 Corpus Analysis

We performed a corpus analysis on existing Java code to assess how frequently there are opportunities to use TSLs. As a lower bound for this metric, we examined `String` arguments passed into Java constructors, for two reasons:

1. The `String` type may be used to represent a large variety of notations, many of which may be expressed using TSLs.
2. We hypothesized that opportunities to use TSLs would often come when instantiating an object.

Methodology. We ran our analysis on a recent version (20130901r) of the Qualitas Corpus [33], consisting of 107 Java projects, and searched for constructors that used `Strings` that could be substituted with TSLs. To perform the search, we used command line tools, such as `grep` and `sed`, and a text editor features such as search and substitution. After we found the constructors, we chose those that took at least one `String` as an argument. Via a visual scan of the names of the constructors and their `String` arguments, we inferred how the constructors and the arguments were intended to be used. Some additional details are provided in the technical report [24].

Results. We found 124,873 constructors and that 19,288 (15%) of them could use TSLs. Table 1 gives more details on types of `String` arguments we found that could be substituted with TSLs. The "Identifier" category comprises process IDs, user IDs, column or row IDs, etc. that usually must be unique; the "Pattern" category includes regular expressions, prefixes and suffixes, delimiters, format templates, etc.; the "Other" category contains `Strings` used for ZIP codes, passwords, queries, IP addresses, versions, HTML and XML code, etc.; and the "Directory path" and "URL/URI" categories are self-explanatory.

Table 1: Types of `String` arguments in Java constructors that could use TSLs

Type of String	**Number**	**Percentage**
Identifier	15,642	81%
Directory path	823	4%
Pattern	495	3%
URL/URI	396	2%
Other (ZIP code, password, query, HTML/XML, IP address, version, etc.)	1,932	10%
Total:	**19,288**	**100%**

Limitations. There are three limitations to our corpus analysis. First, the proxy that we chose for finding how often TSLs could be used in existing Java code is imprecise. Our corpus analysis focused exclusively on Java constructors and thus did not consider other programming constructs, such as method calls, assignments, etc., that could possibly use TSLs. We did not count types that themselves could have a TSL associated with them (e.g. `URL`), only uses of `Strings` that we hypothesized might not have been `Strings` had better syntax been available. Our search for constructors with the use of command line tools and text editor features may not have identified every Java constructors present in the corpus. Finally, the inference of the intended functionality of the constructor and the passed in `String` argument was based on the authors' programming experience and was thus subjective.

Despite the limitations of our corpus analysis, it shows that there are many potential use cases where type-specific languages could be considered, given that numerous `String` arguments appeared to specify a parseable format.

6 Implementation

Because Wyvern itself is an evolving language and we believe that the techniques herein are broadly applicable, we have implemented the abstract syntax, typechecking and elaboration rules precisely as specified in this paper, including the hygiene mechanism, in Scala as a stable resource. We have also included a simple compiler from our representation of internal terms, which includes explicit type information at each node, to Scala source code. We represent both external terms and translational terms using the same case classes, using traits to distinguish them when necessary. This code can be used to better understand the implementation overhead of our mechanisms. The key "trick" is to make sure that the typing context also maps each source variable to a unique internal variable, so that elaboration of spliced terms is capture-avoiding. This code can be found at `http://github.com/wyvernlang/tslwyvern`.

Wyvern itself also supports a variant of this mechanism. The Wyvern language is an evolving effort involving a number of techniques other than TSLs, so the implementation does not precisely coincide with the specification presented herein. In particular, Wyvern's object types and case types have substantially different semantics. Moreover, Adams grammars do not presently have a robust implementation, so their presentation here is merely expository. The top-level parser for Wyvern is instead produced by the Copper parser generator [36] which uses stateful LALR parsing to handle whitespace. Forward references, such as the TSL tilde, the new keyword, and case expressions, are handled by inserting a special "signal" token into the parse stream at the end of an expression containing a forward reference. When the parser subsequently reads this signal token, it enters the appropriate state depending on the type of forward reference encountered. TSL blocks are handled as if they were strings, preserving all non-leading whitespace, and new and case expression bodies are parsed using their respective grammars. Wyvern performs literal parsing during typechecking essentially as described, using a standard bidirectional type system. It does not enforce the constraints on parse streams and the hygiene mechanisms as of this writing. Some of the API is implemented using a Java interoperability layer rather than directly in Wyvern. This implementation does support some simpler examples fully, however (unlike the implementation above, which does not have a concrete syntax at all). The code can be found at `http://github.com/wyvernlang/wyvern`.

7 Related Work

Closely related to our approach of type-driven parsing is a concurrent paper by Ichikawa et al. [11] that presents *protean operators*. The paper describes the *ProteaJ* language, based on Java, which allows a programmer to define flexible operators annotated with named types. Syntactic conflict is resolved by looking at the expected type. Conflicts may still arise when the expected type matches two protean operators; in this case ProteaJ allows the programmer to explicitly disambiguate, as in other systems. In contrast, by associating parsers with types, our approach avoids all conflicts, achieving a stricter notion of modularity at the cost of some expressiveness (we only consider delimited literals – these may define operators inside, but we cannot support custom operator syntax directly at the top level). We also give a type theoretic foundation for our approach.

Another way to approach language extensibility is to go a level of abstraction above parsing, as is done via metaprogramming and macro facilities, with Scheme and other Lisp-style languages' hygienic macros being the 'gold standard' for hygiene. In those languages, macros are written in the language itself and use its simple syntax – parentheses universally serve as expression delimiters (although proposals for whitespace as a substitute for parentheses have been made [21]). Our work is inspired by this flexibility, but aims to support richer syntax as well as maintain a static type discipline. Wyvern's use of types to trigger parsing avoids the overhead of invoking macros explicitly by name, and makes it easier to compose TSLs declaratively. Static macro systems also exist. For instance, OJ (previously, OpenJava) [32] provides a macro system based on a meta-object protocol, and Backstage Java [27], Template Haskell [30] and Converge [34] also employ compile-time meta-programming, the latter with some support for whitespace delimited blocks. Each of these systems provide macro-style rewriting of source code, but they provide at most limited extension of language parsing. String literals can be reinterpreted, but splicing is not hygienic if this is done.

Other systems aim at providing forms of syntax extension that change the host language, as opposed to our whitespace-delimited approach. For example, Camlp4 [4] is a preprocessor for OCaml that can be used to extend the concrete syntax of the language with parsers and extensible grammars. SugarJ [6] supports syntactic extension of the Java language by adding libraries. Wyvern differs from these approach in that the core language is not extended directly, so conflicts cannot arise at link-time.

Scoping TSLs to expressions of a single type comes at the expense of some flexibility, but we believe that many uses of domain-specific languages are of this form already. A previous approach has considered type-based disambiguation of parse forests for supporting quotation and anti-quotation of arbitrary object languages [2]. Our work is similar in spirit, but does not rely on generation of parse forests and associates grammars with types, rather than types with grammar productions. This provides stronger modularity guarantees and is arguably simpler. C# expression trees [19] are similar in that, when the type of a term is, e.g., `Expression<T->T'>`, it is parsed as a quotation. However, like the work just mentioned, this is *specifically* to support quotations. Our work supports quotations as one use case amongst many.

Many approaches to syntax extension, such as XJ [3] are keyword-delimited in some form. We believe that a type-directed approach is more seamless and natural, coinciding with how one would build in language support directly. These approaches also differ in that they either do not support hygienic expansion, or have not specified it in the simple manner that we have.

In terms of work on safe language composition, Schwerdfeger and van Wyk [29] proposed a solution that make strong safety guarantees provided that the languages comply with certain grammar restrictions, concerning first and follow sets of the host language and the added new languages. It also relied on strongly named entry tokens, as with keyword delimited approaches. Our approach does not impose any such restrictions while still making safety guarantees.

Domain-specific language frameworks and language workbenches, such as Spoofax [14], Ensō [18] and others [35], also provide a possible solution for the language extension task. They provide support for generating new programming languages and tooling in a modular manner. The Marco language [16] similarly provides macro definition at a level of abstraction that is largely independent of the target language. In these approaches, each TSL is *external* relative to the host language; in contrast, Wyvern focuses on *internal* extensibility, improving interoperability and composability.

Ongoing work on projectional editors (e.g., [12, 5]) uses a special graphical user interface to allow the developer to implicitly mark where the extensions are placed in the code, essentially directly specifying the underlying ASTs. This solution to the language extension problem is of considerable interest to us, but remains relatively understudied formally. It is likely that a type-oriented approach to projectional editing, inspired by that described herein, could be fruitful.

We were informed by our previous work on Active Code Completion (ACC), which associates code completion palettes with types [25], much as we associate parsers with types. ACC palettes could be used for defining a TSL syntax for types in a complementary manner. In ACC that syntax is immediately translated to Java syntax at edit time, while this work integrates with the language, so the syntax is retained with the code. ACC supports more general interaction modes than just textual syntax, situated between our approach and projectional editors.

8 Discussion

We have presented a minimal but complete language design that we believe is particularly elegant, practical and theoretically well-motivated. The key to this is our organization of language extensions around types, rather than around grammar fragments.

There are several directions that remain to be explored:

- TSL Wyvern does not support polymorphic types, like `'a list` in our first example. Were we to add support for them, we would expect that the type constructor (`list`) would determine the syntax, not the particular type. Thus, we may fundamentally be proposing *type constructor specific languages*.
- Similarly, TSL Wyvern does not support abstract types. It may be useful to include the ability to associate metadata with an abstract type, much in the same way that we associate metadata with a named type here.
- TSLs as described here allow one to give an alternative syntax for introductory term forms, but elimination forms cannot be defined directly. There are two directions we may wish to go to support this:
  1. Pattern matching is a powerful feature supported by an increasing number of languages. Pattern syntax is similar to term syntax. It may be possible for a TSL definition to include parse functions for “literal-like” forms appearing in patterns, elaborating them to pattern terms rather than expression terms.
  2. Keywords are more useful when defining custom elimination forms (e.g. `if` based on `case`). It may be possible to support “typed syntax macros” using the same hygiene mechanisms we described here.
- We do not provide TSLs with the ability to diverge based on the type of a spliced expression. This might be useful if, for example, our HTML TSL wanted to treat spliced strings differently from other spliced HTML terms. For polymorphic types, we might also wish to diverge based on the type index.

- We may wish to design less restrictive shadowing constraints, so that TSLs can introduce variables directly into the scope of a spliced expression if they explicitly wish to (bypassing the need for the client to provide a function for the TSL to call). The community may wish to discuss whether this is worth the cost in terms of difficulty of determining where a variable has been bound.
- We need to provide further empirical validation. This may benefit from the integration of TSLs into existing languages other than Wyvern.
- We need to consider broader IDE support – custom syntax benefits from custom editor support, and it may be possible to design IDEs that dispatch to type metadata in much the way the typechecker does in this paper. Our informal considerations of existing IDE extension mechanisms suggests that this may be non-trivial.

Acknowledgements

We thank the anonymous reviewers, Joshua Sunshine, Filipe Militão and Eric Van Wyk for helpful comments and discussions, and acknowledge the support of the United States Air Force Research Laboratory and the National Security Agency lablet contract #H98230-14-C-0140, as well as the Royal Society of New Zealand Marsden Fund. Cyrus Omar was supported by an NSF Graduate Research Fellowship.

References

1. M. D. Adams. Principled parsing for indentation-sensitive languages: Revisiting Landin's offside rule. In *Principles of Programming Languages*, 2013.
2. M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Generative Programming and Component Engineering*, 2005.
3. T. Clark, P. Sammut, and J. S. Willans. Beyond annotations: A proposal for extensible Java (XJ). In *Source Code Analysis and Manipulation*, 2008.
4. D. de Rauglaudre. Camlp4 - Reference Manual. `http://caml.inria.fr/pub/docs/manual-camlp4/`, 2003.
5. L. Diekmann and L. Tratt. Parsing composed grammars with language boxes. In *Workshop on Scalable Language Specification*, 2013.
6. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based language extensibility. In *Object-Oriented Programming Systems, Languages, and Applications*, 2011.
7. S. Erdweg and F. Rieger. A framework for extensible languages. In *Generative Programming: Concepts & Experiences*, 2013.
8. T. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.
9. R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.
10. R. Harper and C. Stone. A Type-Theoretic Interpretation of Standard ML. In *IN Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
11. K. Ichikawa and S. Chiba. Composable user-defined operators that can express user-defined literals. In *Modularity*, 2014.
12. JetBrains. JetBrains MPS – Meta Programming System. `http://www.jetbrains.com/mps/`.
13. V. Karakoidas. On domain-specific languages usage (why DSLs really matter). *Crossroads*, 20(3):16–17, Mar. 2014.

14. L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Object-Oriented Programming Systems, Languages, and Applications*, 2010.
15. L. Krishnan and E. Van Wyk. Termination analysis for higher-order attribute grammars. In *Software Language Engineering*, 2012.
16. B. Lee, R. Grimm, M. Hirzel, and K. S. McKinley. Marco: Safe, expressive macros for any language. In *European Conference on Object-Oriented Programming*, 2012.
17. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 2009.
18. A. Loh, T. van der Storm, and W. R. Cook. Managed data: Modular strategies for data abstraction. In *Onward!*, 2012.
19. Microsoft Corporation. Expression Trees (C# and Visual Basic). `http://msdn.microsoft.com/en-us/library/bb397951.aspx`.
20. H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Object Oriented Programming Systems, Languages & Applications*, 2013.
21. E. Möller. SRFI-49: Indentation-sensitive syntax. `http://srfi.schemers.org/srfi-49/srfi-49.html`, 2005.
22. L. Nistor, D. Kurilova, S. Balzer, B. Chung, A. Potanin, and J. Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *MechAnisms for SPEcialization, Generalization and inHerItance*, 2013.
23. C. Omar, B. Chung, D. Kurilova, A. Potanin, and J. Aldrich. Type-directed, whitespace-delimited parsing for embedded DSLs. In *Globalization of Domain Specific Languages*, 2013.
24. C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely Composable Type-Specific Languages. Technical Report CMU-ISR-14-106, Carnegie Mellon University, 2014.
25. C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *International Conference on Software Engineering*, 2012.
26. OWASP. OWASP Top 10 2013. `https://www.owasp.org/index.php/Top_10_2013-Top_10`, 2013.
27. Z. Palmer and S. F. Smith. Backstage Java: Making a Difference in Metaprogramming. In *Object-Oriented Programming Systems, Languages, and Applications*, 2011.
28. B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.
29. A. C. Schwerdfeger and E. R. Van Wyk. Verifiable composition of deterministic grammars. In *Programming Language Design and Implementation*, 2009.
30. T. Sheard and S. Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.
31. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.
32. M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. In *Reflection and Software Engineering*, 2000.
33. E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference*, 2010.
34. L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6), Oct. 2008.
35. M. G. J. van den Brand. *Pregmatic: A Generator for Incremental Programming Environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.
36. E. R. Van Wyk and A. C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Generative programming and component engineering*, 2007.

Rely-Guarantee Protocols

Filipe Militão[1,2], Jonathan Aldrich[1], and Luís Caires[2]

[1] Carnegie Mellon University, Pittsburgh, USA
[2] Universidade Nova de Lisboa, Lisboa, Portugal
{filipe.militao,jonathan.aldrich}@cs.cmu.edu lcaires@fct.unl.pt

Abstract. The use of shared mutable state, commonly seen in object-oriented systems, is often problematic due to the potential conflicting interactions between aliases to the same state. We present a substructural type system outfitted with a novel lightweight interference control mechanism, *rely-guarantee protocols*, that enables controlled aliasing of shared resources. By assigning each alias separate roles, encoded in a novel protocol abstraction in the spirit of rely-guarantee reasoning, our type system ensures that challenging uses of shared state will never interfere in an unsafe fashion. In particular, rely-guarantee protocols ensure that each alias will never observe an unexpected value, or type, when inspecting shared memory regardless of how the changes to that shared state (originating from potentially unknown program contexts) are interleaved at run-time.

1 Introduction

Shared, mutable state can be useful in certain algorithms, in modeling stateful systems, and in structuring programs. However, it can also make reasoning about a program more difficult, potentially resulting in run-time errors. If two pieces of code have references to the same location in memory, and one of them updates the contents of that cell, the update may *destructively interfere* by breaking the other piece of code's assumptions about the properties of the value contained in that cell—which may cause the program to compute the wrong result, or even to abruptly terminate. In order to mitigate this problem, static type systems conservatively associate an invariant type with each location, and ensure that every store to the location preserves this type. While this approach can ensure basic memory safety, it cannot check higher-level *protocol* properties [1,4,5,13,20] that are vital to the correctness of many programs [3].

For example, consider a `Pipe` abstraction that is used to communicate between two parts of the program. A pipe is *open* while the communication is ongoing, but when the pipe is no longer needed it is *closed*. Pipes include shared, mutable state in the form of an internal buffer, and abstractions such as Java's `PipedInputStream` also dynamically track whether they are in the open or closed state. The state of the pipe determines what operations may be performed, and invoking an inappropriate operation is an error: for example, writing to a closed pipe in Java results in a run-time exception.

Static approaches to reason about such state protocols (of which we follow the *typestate* [7,22,28,29] approach) have two advantages: errors such as writing to a closed pipe can be avoided on the one hand, and defensive run-time tests of the state of an object can become superfluous on the other hand. In typestate systems, abstractions expose a more refined type that models a set of abstract states representing the internal, changing, type of the state (such as the two states above, *open* and *closed*) enabling the static modular manipulation of stateful objects. However, sharing (such as by aliasing) these resources must be carefully controlled to avoid potentially destructive interference that may result from mixing incompatible changes to apparently unrelated objects that, in reality, are connected to the same underlying run-time object. This work aims to provide an intuitive and general-purpose extension to the typestate model by exploiting (coordination) protocols at the shared state level to allow fine-grained and flexible uses of aliased state. Therefore, by modeling the *interactions* of aliases of some shared state in a protocol abstraction, we enable complex uses of sharing to safely occur through *benign interference*, interference that the other aliases expect and/or require to occur.

Consider once more the pipe example. The next two code blocks implement simplified versions of the pipe's `put` and `tryTake` functions. Although each function operates independently of the other, internally they share nodes of the same underlying buffer:

```
// protocol: Empty ⇒ Filled; none
put = fun( v : Value ).
  // Empty shared node, oldlast, to be filled with node
  // containing tagged (#) empty record, {}, as ‘Empty’
  let last = new Empty#{} in
    let oldlast = !buffer.tail in // is Empty
      // tags pair of ‘v’ and ‘last’ as ‘Filled’
      oldlast := Filled#{ v , last };
      buffer.tail := last
    end // last cell is now reachable from head&tail
  end // oldlast cell unreachable from tail
```

```
// rec X.( Empty ⇒ Empty; X ⊕ Filled ⇒ none )
tryTake = fun().
  let first = !buffer.head in
    case !first of
      Empty#_ → NoResult#{}
    | Filled#[ v , next ] → // does not return
        delete first; // ownership to the protocol
        buffer.head := next;
        Result#v
    end
  end
```

By distributing these functions between two aliases, we are able to create independent *producer* and *consumer* components of the pipe that share a common buffer (modeled as a singly-linked list). Observe how the interaction, that occurs through aliases of the buffer's nodes, obeys a well-defined protocol: the *producer* alias (through the `put` function) inserts an element into the last (empty) node of the buffer and then immediately forfeits that cell (i.e. it is no longer used by that alias); while the *consumer* alias (using `tryTake`) proceeds by testing the first node and, when it detects it has been `Filled` (thus, when the other alias is sure to no longer use it), recovers ownership of that node, which enables the alias to safely delete that cell (`first`) since it is no longer shared.

1.1 Approach in a Nutshell

Interference due to aliasing is analogous to the interference caused by thread interleaving [15,33]. This occurs because mutable state may be shared by aliases in unknown or non-local program contexts. Such boundary effectively negates the use of static mechanisms to track exactly which other variables alias some state. Therefore, we are unable to know precisely if the shared state aliased by a local variable will be used when the execution jumps off (e.g. through a function call) to non-local program contexts. However, if that state is used, then the aliases may change the state in ways that invalidate the local alias' assumptions on the current contents of the shared state. This interference caused by "alias interleaving" occurs even without concurrency, but is analogous to how thread interleaving may affect shared state. Consequently, techniques to reason about thread interference (such as *rely-guarantee reasoning* [17]) can be useful to reason about aliasing even in our sequential setting. The core principle of rely-guarantee reasoning that we adapt is its mechanism to make strong local assumptions in the face of interference. To handle such interference, each alias has its actions constrained to fit within a *guarantee* type and at the same time is free to assume that the changes done by other aliases of that state must fit within a *rely* type. The duality between what aliases can rely on and must guarantee among themselves yields significant flexibility in the use of shared state, when compared for instance to invariant-based sharing.

We employ rely-guarantee in a novel protocol abstraction that captures a partial view of the use of the shared state, as seen from the perspective of an alias. Therefore, each protocol models the constraints on the actions of that alias and is only aware of the resulting effects ("interference") that may appear in the shared state due to the interleaved uses of that shared state as done by other aliases. A rely-guarantee protocol is formed by a sequence of rely-guarantee steps. Each step contains a rely type, stating what an alias currently assumes the shared state contains; and a guarantee type, a promise that the changes done by that alias will fit within this type. Using these small building blocks, our technique allows strong local assumption on how the shared state may change, while not knowing when or if other aliases to that shared state will be used—only how they will interact with the shared state, if used. Since each step in a protocol can have distinct rely and guarantee types, a protocol is not frozen in time and can model different "temporal" uses of the shared state directly. A protocol is, therefore, an abstracted perspective on the actions done by each individual alias to the shared state, and that is only aware of the potential resulting effects of all the other aliases of that shared state. A *protocol conformance* mechanism ensures the sound composition of all protocols to the same shared state, at the moment of their creation. From there on, each protocol is *stable* (i.e. immune to unexpected/destructive interference) since conformance attested that each protocol, in isolation, is aware of all observable effects that may occur from all possible "alias interleaving" originated from the remaining aliases.

Our main contribution is a novel type-based protocol abstraction to reason about shared mutable state, *rely-guarantee protocols*, that captures the following features:

1. Each protocol provides a *local* type so that an alias need not know the actions that other aliases are doing, only their resulting (observable) effect on the shared state;
2. Sharing can be done *asymmetrically* so that the role of each alias in the interaction with the shared state may be distinct from the rest;
3. Our protocol paradigm is able to *scale* by modeling sharing interactions both at the reference level and also at the abstract state level. Therefore, sharing does not need to be embedded in an ADT [18], but can also work at the ADT level without requiring a wrapper reference [15];
4. State can be shared individually or simultaneously in groups of state. By enabling sharing to occur underneath a layer of apparently disjoint state, we naturally support the notion of *fictional disjointness* [9, 16, 18];
5. Our protocol abstraction is able to model complex interactions that occur through the shared state. These include invariant, monotonic and other coordinated uses. Moreover, they enable both *ownership transfer* of state between non-local program contexts and *ownership recovery*. Therefore, shared state can return to be non-shared, even allowing it to be later shared again and in such a way that is completely unrelated to its previous sharing phases;

6. Although protocol conformance is checked in pairs, *arbitrary aliasing* is possible (if safe) by further sharing a protocol in ways that do not conflict with the initial sharing. Therefore, global conformance in the use of the shared state by multiple aliases is assured by the combination of individual binary protocol splits, with each split sharing the state without breaking what was previously assumed on that state;
7. We allow *temporary inconsistencies*, so that the shared state may undergo intermediate (private) states that cannot be seen by other aliases. Using an idea similar to (static) mutual exclusion, we ensure that the same shared state cannot be inspected while it is inconsistent. Such kind of critical section (that does not incur in any run-time overhead) is sufficiently flexible to support multiple simultaneously inconsistent states, when they are sure to not be aliasing the same shared state.

With this technique we are able to model challenging uses of aliasing in a lightweight substructural type system, where all sharing is centered on a simple and intuitive protocol abstraction. We believe that by specializing our system to typestate and aliasing [1, 27] properties we can offer a useful intermediate point that is simpler than the full functional verification embodied in separation logic [6,25] yet more expressive than conventional type systems. Our proofs of soundness use standard progress and preservation theorems. We show that all allowed interference is benign (i.e. that all changes to the shared state are expected by each alias) by ensuring that a program cannot get stuck, while still allowing the shared state to be legally used in complex ways. Besides expressing the programmer's intent in the types, our technique also enables a program to be free of errors related to destructive interference. For instance, the programmer will not be able to wrongly attempt to use a shared cell as if it were no longer shared, or leave values in that shared cell that are not expected by the other aliases of that cell.

Section 2 introduces the language but leaves its sharing mechanisms to Section 4, after an overview of the type system. Section 5 discusses technical results, and Section 6 additional examples. The paper ends with Sections for related work and conclusions.

2 Pipe Example

Our language is based on the polymorphic λ-calculus with mutable references, immutable records, tagged sums and recursive types. Technically, we build on [22] (a variant of $\mathbf{L}^3$ [1] adapted for usability) by supporting sharing of mutable state through rely-guarantee protocols. As in $\mathbf{L}^3$, a cell is decomposed in two components: a pure *reference* (that can be freely copied), and a linear [14] *capability* used to track the contents of that cell. Unlike $\mathbf{L}^3$, by extending [22] our language implicitly threads capabilities through the code, reducing syntactic overhead. To support this separation of references and capabilities, our language uses location-dependent types to relate a reference to its respective capability. Therefore, a reference has a type "**ref** *t*" to mean a **ref**erence to a location *t*, where the information about the contents of that location is stored in the capability for *t*. Our capabilities follow the format "**rw** *t A*" meaning a **r**ead-**w**rite capability to location *t* which, currently, has contents of type *A* stored in it. The permission to access, such as by dereference, the contents of a cell requires both the reference and the capability to be available. Capabilities are typing artifacts that do not exist at run-time and are moved implicitly through the code. Locations (such as t) must be managed explicitly, leading to constructs dedicated to abstracting and opening locations.

Pipes are used to support a *consumer-producer* style of interaction (using a shared internal buffer as mediator), often used in a concurrent program but here used in a single-threaded environment. The shared internal buffer is implemented as a shared singly-linked list where the consumer keeps a pointer to the *head* of the list and the producer to its *tail*. By partitioning the pipe's functions (where the consumer alias uses `tryTake`, and the producer both `put` and `close`), clients of the pipe can work independently of one another, provided that the functions' implementation is aware of the potential interference caused by the actions of the other alias. It is on specifying and verifying this interference that our rely-guarantee protocols will be used.

```
 1 let newPipe = fun( _ : [] ).                                          Γ = _ : [] | Δ = ·
 2   open <n,node> = new Empty#{} in                   Γ = _ : [], node : ref n, n : loc | Δ = rw n Empty#[]
 3    share (rw n Empty#[]) as H[n] || T[n];                                 Γ = ... | Δ = T[n], H[n]
 4    open <h,head> = new <n, node::H[n]> in Γ = ..., head : ref h, h : loc | Δ = T[n], rw h ∃p.(ref p :: H[p])
 5     open <t,tail> = new <n, node::T[n]> in   Γ = ..., tail : ref t, t : loc | Δ = rw t ∃p.(ref p :: T[p]), ...
 6     < rw h exists p.(ref p :: H[p]), // packs a type, the capability to location 'h'
 7     < rw t exists p.(ref p :: T[p]), // packs a type, the capability to location 't'
 8     { // creates labeled record with 'put', 'close' and 'tryTake' as members
 9      put = fun( e : int :: rw t exists p.(ref p :: T[p]) )./*...shown in Section 4...*/,
19      close = fun( _ : [] :: rw t exists p.(ref p :: T[p]) )./*...*/,
26      tryTake = fun( _ : [] :: rw h exists p.(ref p :: H[p]) )./*...*/
47     } :: ( rw h exists p.(ref p :: H[p]) * rw t exists p.(ref p :: T[p]) ) > >
48     end
49    end
50   end
```

The function creates a pipe by allocating an initial `node` for the internal buffer, a cell to be shared by the `head` and `tail` pointers. The newly allocated cell (line 2) contains a tagged (as `Empty`) empty record ({}). In our language, aliasing information is correlated through static names, *locations*, such that multiple references to the same location must imply that these references are aliases of the same cell. Consequently, the new construct (line 2) must be assigned a type that abstracts the concrete location that was created, $\exists t.(\ \textbf{ref}\ t :: \textbf{rw}\ t\ \texttt{Empty\#[]}\ )$, which means that there exists some fresh location t, and the new expression evaluates to a reference to t ("**ref** t"). We associate this reference with a capability to access it, using a *stacking* operator ::. In this case the capability is **rw** t `Empty#[]`, representing a **r**ead and **w**rite capability to the location t, which currently contains a value of type `Empty#[]` as initially mentioned. On the same line, we then open the existential by giving it a location variable n and a regular variable `node` to refer that reference. From there on, the capability (a typing artifact which has no actual value) is automatically *unstacked* and moved implicitly as needed through the program. For clarity, we will manually stack capabilities (such as on line 4, using the construct $e :: A$ where A is the stacked capability), although the type system does not require it. On line 3, the type system initially carries the following assumptions:

$$\Gamma \ = \ \_ : [\,]\ ,\ \texttt{node} : \textbf{ref}\ n\ ,\ n : \textbf{loc} \qquad | \qquad \Delta \ = \ \textbf{rw}\ n\ \texttt{Empty\#[]}$$

where Γ is the lexical environment (of persistent/pure resources), and Δ is a linear typing environment that contains all linear resources (such as capabilities). Each linear capability must either be used up or passed on through the program (e.g. by returning it from a function). The contents of the reference `node` are known statically by looking up the capability for the **loc**ation n to which `node` refers (i.e. "**rw** n `Empty#[]`").

Capabilities are linear (cannot be duplicated), but aliasing in local contexts is still possible by copying references. All copies link back to the same capability using the location contained in the reference. However, when aliases operate in non-local contexts, this location-based link is lost. Thus, if we were to pack `node`'s capability before sharing it, it would become unavailable to other aliases of that location. For instance, by writing $\langle n, \texttt{node} :: \textbf{rw}\ n\ \texttt{Empty\#}[] \rangle$ we pack the location n by abstracting it in an existential type for that location. The packed type now refers a fresh location, unrelated to its old version. Instead, we share that capability (line 3) by splitting it in two rely-guarantee protocols, H and T [3]. Each protocol is then assigned to the `head` and `tail` pointers (lines 4 and 5, respectively), since they encode the specific uses of each of those aliases. The protocols and sharing mechanisms will be introduced in Section 4.

The type of `newPipe` is a linear function ($\multimap$) that, since it does not capture any enclosing linear resource, can be marked as pure (!) so that the type can be used without the linear restriction. On line 6 we pack the inner state of the pipe (so as to abstract the capability for t as $\texttt{P}$, and the one for h as $\texttt{C}$), resulting in `newPipe` having the type:

$$\texttt{newPipe}\ :\ !(\ []\ \multimap\ \exists \texttt{C}.\exists \texttt{P}.(\ ![\ ...\ ]\ ::\ \texttt{C} * \texttt{P}\ )\ )$$

where the *separate* capabilities for the Consumer and Producer are stacked together in a commutative group ($*$). In this type, $\texttt{C}$ abstracts the capability $\textbf{rw}\ h\ \exists p.(\textbf{ref}\ p :: \mathsf{H}[p])$, and $\texttt{P}$ abstracts $\textbf{rw}\ t\ \exists p.(\textbf{ref}\ p :: \mathsf{T}[p])$. Finally, although we have not yet shown the implementation, the type of the elided record ([...]) contains function types that should be unsurprising noting that each argument and return type has the respective capabilities for the `head`/`tail` cells stacked on top (similarly to pre/post conditions, but directly expressed in the types). Therefore, those functions are closures that use the knowledge about the reference to the `head`/`tail` pointers from the surrounding context, but do not capture the capability to those cells and instead require them to be supplied as argument.

$$\begin{array}{lll}
[\ \mathsf{put} & : & !(\ \mathsf{int} :: \texttt{P} \multimap []\ ::\ \texttt{P}\ ), \\
\ \ \mathsf{close} & : & !(\ []\ ::\ \texttt{P} \multimap []\ ), \\
\ \ \mathsf{tryTake} & : & !(\ []\ ::\ \texttt{C} \multimap \texttt{NoResult\#}([]\ ::\ \texttt{C}) + \texttt{Result\#}(\mathsf{int} :: \texttt{C}) + \texttt{Depleted\#}[]\ )\ ]
\end{array}$$

Therefore, `put` preserves the producer's capability, but `close` destroys it; while the result of `tryTake` is a sum type of either `Result` or `NoResult` depending on whether the still open pipe has or not contents available, or `Depleted` to signal that the pipe was closed (and therefore that the capability to `C` vanished). Observe that the state that the functions depend on is, apparently, disjoint although underneath this layer the state is actually shared (but coordinated through a protocol) so that (benign) interference must occur for the pipe to work properly—i.e. it is *fictionally disjoint* [9, 16, 18].

3 Type System Overview

We now present the type system. Non-essential details are relegated to [21, 22]. For consistency, we include all sharing mechanisms but leave their discussion to Section 4.

[3] As a brief glimpse, T is "$\textbf{rw}\ n\ \texttt{Empty\#}[]\ \Rightarrow\ (\ \textbf{rw}\ n\ \texttt{Node\#}\mathsf{R}\ \oplus\ \textbf{rw}\ n\ \texttt{Closed\#}[]\ ); \textbf{none}$" which relies on n containing $\texttt{Empty\#}[]$, ensures n then contains either $\texttt{Node\#}\mathsf{R}$ or $\texttt{Closed\#}[]$, and then loses access to n. Both "$\Rightarrow$" and ";" (and R) will be discussed in detail in Section 4.

$$\rho \;\in\; \textsc{Location Constants (Addresses)} \qquad t \;\in\; \textsc{Location Variables} \qquad p \;::=\; \rho \mid t$$

$$\texttt{l} \;\in\; \textsc{Labels (Tags)} \qquad \texttt{f} \;\in\; \textsc{Fields} \qquad x \;\in\; \textsc{Variables} \qquad X \;\in\; \textsc{Type Variables}$$

$$\begin{array}{lll lll}
v ::= & \rho & \text{(address)} & \mid & v.\texttt{f} & \text{(field)} \\
\mid & x & \text{(variable)} & \mid & v\ v & \text{(application)} \\
\mid & \mathsf{fun}(x : A).e & \text{(function)} & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end} & \text{(let)} \\
\mid & \langle t \rangle\, e & \text{(universal location)} & \mid & \mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} & \text{(open location)} \\
\mid & \langle X \rangle\, e & \text{(universal type)} & \mid & \mathsf{open}\ \langle X, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} & \text{(open type)} \\
\mid & \langle p, v \rangle & \text{(pack location)} & \mid & \mathsf{new}\ v & \text{(cell creation)} \\
\mid & \langle A, v \rangle & \text{(pack type)} & \mid & \mathsf{delete}\ v & \text{(cell deletion)} \\
\mid & \{\overline{\texttt{f} = v}\} & \text{(record)} & \mid & !v & \text{(dereference)} \\
\mid & \texttt{l}\#v & \text{(tagged value)} & \mid & v := v & \text{(assign)} \\
 & & & \mid & \mathsf{case}\ v\ \mathsf{of}\ \overline{\texttt{l}\#x \rightarrow e}\ \mathsf{end} & \text{(case)} \\
e ::= & v & \text{(value)} & \mid & \mathsf{share}\ A_0\ \mathsf{as}\ A_1 \parallel A_2 & \text{(share)} \\
\mid & v[p] & \text{(location application)} & \mid & \mathsf{focus}\ \overline{A} & \text{(focus)} \\
\mid & v[A] & \text{(type application)} & \mid & \mathsf{defocus} & \text{(defocus)}
\end{array}$$

Note: ρ is not source-level. $\overline{Z}$ for a possibly empty sequence of Z. Tuples, recursion, etc. are encoded as idioms, see [22].

Fig. 1. Values (v) and expressions (e).

The (let-expanded [26]) grammar is shown in Fig. 1. The main deviations from standard λ-calculus are the inclusion of location-related constructs, and the sharing constructs (share, focus and defocus).

We use a flat type grammar (Fig. 2) where both capabilities (i.e. typing artifacts without values, which includes our rely-guarantee protocols) and standard types (used to type values) coexist. Our design does not need to make a syntactic distinction between the two kinds since the type system ensures the proper separation in their use. We now overview the basic types, leaving the rely and guarantee types to be presented in the following Section together with the discussion on sharing. Pure types $!A$ enable a linear type to be used multiple times. $A \multimap A'$ describes a linear function of argument A and result A'. The stacking operation $A :: A'$ stacks A' (a capability, or abstracted capability) on top of A. This stacking is not commutative since it stacks a single type on the right of ::. Therefore, $*$ enables multiple types to be grouped together that, when later stacked, allow that type to list a commutative group of capabilities[4]. Both $\forall$ and $\exists$ offer the standard quantification, over location and type kinds, together with the respective location/type variables. $[\overline{\texttt{f} : A}]$ are used to described labeled records of arbitrary length. A **ref** p type is a reference for location p noting that the contents of such a reference are tracked by the capability to that location and not immediately stored in the reference type. **rec**ursive types, that are automatically folded/unfolded through subtyping rules (see Fig. 4 and (t:Subsumption) on Fig. 3), are also supported. Sum types use the form `tag`$\#A$ to tag type A with `tag`. Alternatives ($\oplus$) model imprecision in the knowledge of the type by listing different possible states it may be in. **none** is the empty capability, while **rw** p A is the read-write capability to location p (a memory cell currently con-

[4] Note that while $A_0 :: (A_1 :: A_2)$ and $A_0 :: (A_2 :: A_1)$ are not (necessarily) subtypes, capability commutation is always possible with $*$ such that $A_0 :: (A_1 * A_2) <:> A_0 :: (A_2 * A_1)$.

$$
\begin{array}{rll@{\qquad}lll}
A ::= & !A & \text{(pure/persistent)} & \mid & \mathbf{ref}\ p & \text{(reference type)} \\
\mid & A \multimap A & \text{(linear function)} & \mid & \mathbf{rec}\ X.A & \text{(recursive type)} \\
\mid & A :: A & \text{(stacking)} & \mid & \sum_i \mathtt{l}_i\#A_i & \text{(tagged sum)} \\
\mid & A * A & \text{(separation)} & \mid & A \oplus A & \text{(alternative)} \\
\mid & [\overline{\mathtt{f} : A}] & \text{(record)} & \mid & A \,\&\, A & \text{(intersection)} \\
\mid & X & \text{(type variable)} & \mid & \mathbf{rw}\ p\ A & \text{(read-write capability to } p\text{)} \\
\mid & \forall X.A & \text{(universal type quantification)} & \mid & \textsf{none} & \text{(empty capability)} \\
\mid & \exists X.A & \text{(existential type quantification)} & \mid & A \Rightarrow A & \text{(rely)} \\
\mid & \forall t.A & \text{(universal location quantification)} & \mid & A ; A & \text{(guarantee)} \\
\mid & \exists t.A & \text{(existential location quantification)} & & &
\end{array}
$$

Note: $\sum_i \mathtt{l}_i\#A_i$ denotes a single tagged type or a sequence of tagged types separated by +, such as "$\mathtt{t}\#A + \mathtt{u}\#B + \mathtt{v}\#C$". Separation, sum, alternative and intersection types are assumed commutative, i.e. without respective subtyping rules.

Fig. 2. Types and capabilities.

taining a value of type A). Finally, an $A\&A'$ type means that the client can choose to use either type A or type A' but not both simultaneously.

Our typing rules use typing judgments of the form: $\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1$ stating that with lexical environment Γ and linear resources Δ_0 we assign the expression e a type A and produce effects that result in Δ_1. The typing environments are as follows:

$$
\begin{array}{rll@{\qquad}rll}
\Gamma ::= & \cdot & \text{(empty)} & \Delta ::= & \cdot & \text{(empty)} \\
\mid & \Gamma,\ x : A & \text{(variable binding)} & \mid & \Delta,\ x : A & \text{(linear binding)} \\
\mid & \Gamma,\ p : \mathbf{loc} & \text{(location variable assertion)} & \mid & \Delta,\ A & \text{(capability/protocol)} \\
\mid & \Gamma,\ X : \mathbf{type} & \text{(type assertion)} & \mid & \Delta^G,\ A_0 ; A_1 \triangleright \Delta & \text{(defocus-guarantee)}
\end{array}
$$

where Δ^G syntactically restricts Δ to not include a defocus-guarantee (a sharing feature, see Section 4.3). Suffices to note that this restriction ensures that defocus-guarantees are nested on the right of $\triangleright$ and that, at each level, there exists only one pending defocus-guarantee. Δ^G is also used to forbid capture of defocus-guarantees by functions and other constructs that can keep part of the linear typing environment for themselves.

The main typing rules are shown in Fig. 3, but the last four typing rules are only discussed in Section 4. All values (which includes functions, tagged values, etc.) have no resulting effect ($\cdot$) since, operationally, they have no pending computations. Allocating a new cell results in a type, $\exists t.(\ \mathbf{ref}\ t :: \mathbf{rw}\ t\ A\ )$, that abstracts the fresh location that was created (t), and includes both a reference to that location and the capability to that location. To associate a value (such as **ref** t) with some capability (such as the capability to access location t), we use a *stacking* operator ::. Naturally, to be able to use the existential location, we must first open that abstraction by giving it a *location variable* to refer the abstracted location, besides the usual variable to refer the contents of the existential type. Reading the content of a cell can be either destructive or not, depending on whether its content is pure (!). If it is linear, then to preserve linearity we must leave the unit type ([]) behind to avoid duplication. By banging the type of a variable binding, we can move it to the linear context which enables the function's typing rule to initially consider all arguments as linear even if they are pure. Functions can only capture a Δ^G linear environment to ensure that they will not hide a pending defocus-guarantee

$\boxed{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}$ **Typing rules, (t:*)**

(t:Ref)
$$\frac{}{\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash \rho : \mathbf{ref}\, \rho \dashv \cdot}$$

(t:Unit)
$$\frac{}{\Gamma \mid \cdot \vdash v : [] \dashv \cdot}$$

(t:Pure-Read)
$$\frac{}{\Gamma, x : A \mid \cdot \vdash x : !A \dashv \cdot}$$

(t:Linear-Read)
$$\frac{}{\Gamma \mid x : A \vdash x : A \dashv \cdot}$$

(t:Pure)
$$\frac{\Gamma \mid \cdot \vdash v : A \dashv \cdot}{\Gamma \mid \cdot \vdash v : !A \dashv \cdot}$$

(t:Pure-Elim)
$$\frac{\Gamma, x : A_0 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : !A_0 \vdash e : A_1 \dashv \Delta_1}$$

(t:Tag)
$$\frac{\Gamma \mid \Delta \vdash v : A \dashv \cdot}{\Gamma \mid \Delta \vdash \mathtt{l}\#v : \mathtt{l}\#A \dashv \cdot}$$

(t:Loc-Pack)
$$\frac{\Gamma \mid \Delta \vdash v : A\{p/t\} \dashv \cdot}{\Gamma \mid \Delta \vdash \langle p, v \rangle : \exists t.A \dashv \cdot}$$

(t:New)
$$\frac{\Gamma \mid \Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{new}\ v : \exists t.(\mathbf{ref}\ t :: \mathbf{rw}\ t\ A) \dashv \Delta_1}$$

(t:Delete)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \exists t.(\mathbf{ref}\ t :: \mathbf{rw}\ t\ A) \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{delete}\ v : \exists t.A \dashv \Delta_1}$$

(t:Function)
$$\frac{\Gamma \mid \Delta^G, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma \mid \Delta^G \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(t:Application)
$$\frac{\Gamma \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash v_1 : A_0 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_2}$$

(t:Dereference-Pure)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ !A}{\Gamma \mid \Delta_0 \vdash !v : !A \dashv \Delta_1, \mathbf{rw}\ p\ !A}$$

(t:Dereference-Linear)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A}{\Gamma \mid \Delta_0 \vdash !v : A \dashv \Delta_1, \mathbf{rw}\ p\ []}$$

(t:Assign)
$$\frac{\begin{array}{c}\Gamma \mid \Delta_0 \vdash v_1 : A_0 \dashv \Delta_1 \\ \Gamma \mid \Delta_1 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1\end{array}}{\Gamma \mid \Delta_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_0}$$

(t:Alternative-Left)
$$\frac{\begin{array}{c}\Gamma \mid \Delta_0, A_0 \vdash e : A_2 \dashv \Delta_1 \\ \Gamma \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1\end{array}}{\Gamma \mid \Delta_0, A_0 \oplus A_1 \vdash e : A_2 \dashv \Delta_1}$$

(t:Intersection-Right)
$$\frac{\begin{array}{c}\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \\ \Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_2\end{array}}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \& A_2}$$

(t:Case)
$$\frac{\begin{array}{c}\Gamma \mid \Delta_0 \vdash v : \sum_i \mathtt{l}_i\#A_i \dashv \Delta_1 \\ \overline{\Gamma \mid \Delta_1, x_i : A_i \vdash e_i : A \dashv \Delta_2} \qquad i \leq j\end{array}}{\Gamma \mid \Delta_0 \vdash \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{l}_j\#x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_2}$$

(t:Loc-App)
$$\frac{\begin{array}{c}p : \mathbf{loc} \in \Gamma \\ \Gamma \mid \Delta_0 \vdash v : \forall t.A \dashv \Delta_1\end{array}}{\Gamma \mid \Delta_0 \vdash v[p] : A\{p/t\} \dashv \Delta_1}$$

(t:Forall-Loc)
$$\frac{\Gamma, t : \mathbf{loc} \mid \Delta^G \vdash e : A \dashv \cdot}{\Gamma \mid \Delta^G \vdash \langle t \rangle\, e : \forall t.A \dashv \cdot}$$

(t:Loc-Open)
$$\frac{\begin{array}{c}\Gamma \mid \Delta_0 \vdash v : \exists t.A_0 \dashv \Delta_1 \\ \Gamma, t : \mathbf{loc} \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2\end{array}}{\Gamma \mid \Delta_0 \vdash \mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} : A_1 \dashv \Delta_2}$$

(t:Let)
$$\frac{\begin{array}{c}\Gamma \mid \Delta_0 \vdash e_0 : A_0 \dashv \Delta_1 \\ \Gamma \mid \Delta_1, x : A_0 \vdash e_1 : A_1 \dashv \Delta_2\end{array}}{\Gamma \mid \Delta_0 \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : A_1 \dashv \Delta_2}$$

(t:Subsumption)
$$\frac{\begin{array}{c}\Delta_0 <: \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash e : A_0 \dashv \Delta_2 \\ A_0 <: A_1 \qquad \Delta_2 <: \Delta_3\end{array}}{\Gamma \mid \Delta_0 \vdash e : A_1 \dashv \Delta_3}$$

(t:Cap-Elim)
$$\frac{\Gamma \mid \Delta_0, x : A_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : A_0 :: A_1 \vdash e : A_2 \dashv \Delta_1}$$

(t:Cap-Stack)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

(t:Cap-Unstack)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}$$

(t:Focus-Rely)
$$\frac{A_0 \in \overline{A}}{\Gamma \mid A_0 \Rightarrow A_1 \vdash \mathsf{focus}\ \overline{A} : [] \dashv A_0, A_1 \triangleright \cdot}$$

(t:Defocus-Guarantee)
$$\frac{}{\Gamma \mid \Delta_0, A_0, A_0; A_1 \triangleright \Delta_1 \vdash \mathsf{defocus} : [] \dashv \Delta_0, A_1, \Delta_1}$$

(t:Frame)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \circledast\!\!- \Delta_2 \vdash e : A \dashv \Delta_1 \circledast\!\!- \Delta_2}$$

(t:Share)
$$\frac{A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \mid \Delta, A_0 \vdash \mathsf{share}\ A_0\ \mathsf{as}\ A_1 \parallel A_2 : [] \dashv \Delta, A_1, A_2}$$

Note: all bounded variables of a construct must be fresh in the respective rule's conclusion.

Fig. 3. Static semantics (selected typing rules, see [21] for the rest).

$\boxed{A_0 <: A_1}$ **Subtyping on types, (ST:*)**

$$\text{(ST:ToLinear)}\ \frac{}{!A <: A} \qquad \text{(ST:Unfold)}\ \frac{}{\textbf{rec}\ X.A <: A\{\textbf{rec}\ X.A/X\}} \qquad \text{(ST:Fold)}\ \frac{}{A\{X/\textbf{rec}\ X.A\} <: \textbf{rec}\ X.A}$$

$$\text{(ST:Rec)}\ \frac{A_0 <: A_1}{\textbf{rec}\ X.A_0 <: \textbf{rec}\ X.A_1} \qquad \text{(ST:Sum)}\ \frac{}{\sum_i \mathtt{l}_i\#A_i <: \mathtt{l}'\#A' + \sum_i \mathtt{l}_i\#A_i} \qquad \text{(ST:Alternative)}\ \frac{}{A_0 <: A_0 \oplus A_1} \qquad \text{(ST:Intersection)}\ \frac{}{A_0 \& A_1 <: A_0}$$

$\boxed{\Delta_0 <: \Delta_1}$ **Subtyping on deltas, (SD:*)**

$$\text{(SD:Star)}\ \frac{}{\Delta, A_0, A_1 <:> \Delta, A_0 * A_1} \qquad \text{(SD:Var)}\ \frac{\Delta_0 <: \Delta_1 \qquad A_0 <: A_1}{\Delta_0, x : A_0 <: \Delta_1, x : A_1} \qquad \text{(SD:Type)}\ \frac{\Delta_0 <: \Delta_1 \qquad A_0 <: A_1}{\Delta_0, A_0 <: \Delta_1, A_1} \qquad \text{(SD:None)}\ \frac{}{\Delta <:> \Delta, \textsf{none}}$$

Fig. 4. Subtyping rules (selected, see [21] for the rest).

(and similarly on ∀ abstractions), since our types do not express such pending operation. *Stacking*, done through (T:CAP-ELIM), (T:CAP-STACK) and (T:CAP-UNSTACK) enables the type system to manage capabilities in a non-syntax directed way, since they have no value nor associated identifier. The (T:CASE) rule allows the set of tags of the value that is to be case analyzed (v) to be *smaller* than those listed in the branches of the case ($i \leq j$). This conditions is safe because it amounts to ignoring the effects of those branches, instead of being overly conservative and having to consider them all. These branches are not necessarily useless since, for instance, they may still be relevant on alternative program states (⊕). (T:ALTERNATIVE-LEFT) expresses that if an expression types with both assumptions, A_0 and A_1, then it works with both alternatives. (T:INTERSECTION-RIGHT) is similar but on the resulting effect of that expression.

Finally, (T:SUBSUMPTION) enables expressions to rely on weaker assumptions while ensuring a stronger result than needed. This rule is supported by subtyping rules (a selection is shown in Fig. 4) that follow the form $A_0 <: A_1$ stating that A_0 is a subtype of A_1, meaning that A_0 can be used wherever A_1 is expected. Similar meaning is used for subtyping on linear typing environments, $\Delta_0 <: \Delta_1$. Among other operations, these rules enable automatic fold/unfold of recursive types, as well as grouping ($*$) of resources.

4 Sharing Mutable State

The goal is to enable reads and writes to a cell through multiple aliases, without requiring the type system to precisely track the link between aliased variables. In other words, the type system is aware that a variable is aliased, but does not know exactly which other variables alias that same state. In this scenario, it is no longer possible to implicitly move capabilities between aliases. Instead, we split the original capability into multiple *protocol* capabilities to that same location, and ensure that these multiple protocols cannot interact in ways that destructively interfere with each other. Such *rely-guarantee* protocol accounts for the effects of other protocols (the *rely*), and limits the actions of this protocol to *guarantee* that they do not contradict the assumptions relied on by other aliases. This allows independent, but constrained, actions on the different protocols to the same shared state without destructive interference. However, it also requires us to leverage additional type mechanisms to ensure safety, namely:

(a) **Hide intermediate states.** A rely-guarantee protocol restricts how aliases can use the shared state. However, we allow such specification to be temporarily broken provided that all unexpected changes are private, invisible to other aliases. Therefore, the type system ensures a kind of static mutual exclusion, a mechanism that provides a "critical section" with the desired level of isolation from other aliases to that same state. Consequently, other shared state that may overlap with the one being inspected simply becomes unavailable while that cell is undergoing private changes. Although this solution is necessarily conservative, we avoid any run-time overhead while preserving many relevant usages. To achieve this, we build on the concept of focus [11] (in a non-lexically scoped style, so that there is also a defocus) clearly delimiting the boundary in the code of where shared state is being inspected. Thus, on focus, all other types that may directly or indirectly see inconsistencies must be temporarily concealed only to reappear when those inconsistencies have been fixed, on defocus.

(b) **Ensure that each individual step of the protocol is obeyed.** In our system, sharing properties are encoded in a protocol composed of several rely-guarantee *steps*. As discussed in the previous paragraph, each step must be guarded by focus since private states should not be visible to other aliases. Consequently, the focus construct serves not only to safeguard from interference by other aliases, but also to move the protocol forward through each of its individual steps. At each such step, the code can assume on entry (focus) that the shared state will be in a given well-defined *rely* state, and must ensure on exit (defocus) that the shared state satisfies a given well-defined *guarantee* state. By characterizing the sequence of actions of each alias with an appropriate protocol, one can make strong local assumptions about how the shared state is used without any explicit dependence on how accesses to other aliases of that shared state are interleaved. This feature is crucial since we cannot know precisely if that same shared state was used between two focus-defocus operations.

4.1 Specifying Rely-Guarantee Protocols

We now detail our rely and guarantee types that are the building blocks of our protocols. To clarify the type structure of our protocols, we define the following sub-grammar of our types syntax (Fig. 2) with the types that may appear in a protocol, P.

$$P \ ::= \ \mathbf{rec}\, X.P \mid X \mid P \oplus P \mid P \,\&\, P \mid A \Rightarrow P \mid A; P \mid \textsf{\textbf{none}}$$

A rely-guarantee protocol is a type of capability (i.e. has no value) consisting of potentially many steps, each of the form $A_C \Rightarrow A_P$. Each such step states that it is safe for the current client to assume that the shared state satisfies A_C and is required to obey the guarantee A_P, usually of the form $A'_C; A'_P$ which in turn requires the client to establish (guarantee) that the shared state satisfies A'_C before allowing the protocol to continue to be used as A'_P. Note that our design constrains the syntactical structure of these protocols through *protocol conformance* (Section 4.2), not in the grammar.

Pipe's protocols We can now define the protocols for the shared list nodes of the pipe's buffer. Each node follows a rely-guarantee protocol that includes three possible tagged states: `Node`, which indicates that a list cell contains some useful data; `Empty`, which indicates that the node will be filled with data by the producer (but does not yet have any data); and finally `Closed`, which indicates that the producer has sent all data through the pipe and no more data will be added (thus, it is the last node of the list).

Remember that the producer component of the pipe has an alias to the tail node of the internal list. Because it is the producer, it can rely on that shared node still being `Empty` (as created) since the consumer component will never be allowed to change that state. The rely-guarantee protocol for the tail alias (for some location p) is as follows:

$$\textbf{rw}\ p\ \texttt{Empty\#[]} \Rightarrow (\ \textbf{rw}\ p\ \texttt{Node\#R} \ \oplus\ \textbf{rw}\ p\ \texttt{Closed\#[]}\ ); \textbf{none}$$

This protocol expresses that the client code can safely assume (on focus) a capability stating that location p initially holds type `Empty#[]`. It then requires the code that uses such state to leave it (on defocus) in one of two possible alternatives ($\oplus$) depending on whether the producer chooses to close the pipe or insert a new element to the buffer. To signal that the node is the last element of the pipe, the producer can just assign it a value of type `Closed#[]`. Insertions are slightly more complicated because that action implies that the tail element of the list will be changed. Therefore, after creating the new node, the producer component will keep an alias of the new tail for itself while leaving the old tail with a type that is to be used by the consumer. In this case, the node is assigned a value of type `Node#R`, where R denotes the type $[\ \mathsf{int}\ ,\ \exists p.(\ \textbf{ref}\ p\ ::\ \mathsf{H}[p]\ )\ ]$ (a pair of an integer and a reference to the next shared node of the buffer, as seen from the head pointer). Regardless of its action, the producer then forfeits any ownership of that state which is modeled by the empty capability (**none**)[5] to signal protocol termination.

We now present the abbreviations H and T, the rely-guarantee protocols that govern the use of the shared state of the pipe as seen by the `head` and `tail` aliases, respectively. Note that since we intend to apply the same protocol over different locations, we use "$Q \triangleq \forall p.A$" as a type definition (Q) where we can apply a location without requiring $\forall$ to be a value, such as location q in $Q[q]$. The T and H types are defined as follows:

$$\begin{aligned} \mathsf{T} &\triangleq \forall p.(\ \mathsf{E} \Rightarrow (\ \mathsf{N} \oplus \mathsf{C}\ )\ ) \\ \mathsf{H} &\triangleq \forall p.(\ \textbf{rec}\ X.(\ \mathsf{N} \Rightarrow \textbf{none}\ \oplus\ \mathsf{C} \Rightarrow \textbf{none}\ \oplus\ \mathsf{E} \Rightarrow \mathsf{E}\ ;\ X\ )\ ) \end{aligned}$$

where N is an abbreviation for a capability that contains a node "**rw** p `Node#R`", C is "**rw** p `Closed#[]`" and E is "**rw** p `Empty#[]`". The T type was presented in the paragraph above, so we can now look in more detail to H. Such a protocol contains three alternatives, each with a different action on the state. If the state is found with an E type (i.e. still `Empty`) the consumer is not to modify such state (i.e., just reestablish E), and can retry again later to check if changes occurred. Observe that the remaining two alternatives have a **none** guarantee. This models the recovery of ownership of that particular node. Since the client is not required to reestablish the capability it relied on, that capability can remain available in that context even after defocus.

Each protocol describes a partial view of the complete use of the shared state. Consequently, ensuring their safety cannot be done alone. In our system, protocols are introduced explicitly through the share construct that declares that a type (in practice limited

[5] We frequently omit the trailing "; **none**" for conciseness.

$\boxed{\langle A, P\rangle \rightarrow \langle A', P'\rangle}$ **Step, (STEP:*)**

$$\text{(STEP:NONE)}\;\frac{}{\langle A, \textbf{none}\rangle \rightarrow \langle A, \textbf{none}\rangle} \qquad \text{(STEP:STEP)}\;\frac{}{\langle A_0, A_0 \Rightarrow A_1; P\rangle \rightarrow \langle A_1, P\rangle}$$

$$\text{(STEP:ALTERNATIVE-P)}\;\frac{\langle A_0, P_0\rangle \rightarrow \langle A_1, P_2\rangle}{\langle A_0, P_0 \oplus P_1\rangle \rightarrow \langle A_1, P_2\rangle} \qquad \text{(STEP:ALTERNATIVE-S)}\;\frac{\langle A_0, P_0\rangle \rightarrow \langle A_2, P_1\rangle \qquad \langle A_1, P_0\rangle \rightarrow \langle A_2, P_1\rangle}{\langle A_0 \oplus A_1, P_0\rangle \rightarrow \langle A_2, P_1\rangle}$$

$$\text{(STEP:SUBSUMPTION)}\;\frac{A_0 <: A_1 \qquad P_0 <: P_1 \qquad \langle A_1, P_1\rangle \rightarrow \langle A_2, P_2\rangle \qquad A_2 <: A_3 \qquad P_2 <: P_3}{\langle A_0, P_0\rangle \rightarrow \langle A_3, P_3\rangle}$$

Fig. 5. Protocol stepping rules.

to capabilities, including protocols) is to be split in two new rely-guarantee protocols. Safety is checked by simulating their actions in order to ensure that they preserve the overall consistency in the use of the shared state, no matter how their actions may be interleaved. Since a rely-guarantee protocol can subsequently continue to be split, this technique does not limit the number of aliases provided that the protocols conform.

4.2 Checking Protocol Splitting

The key principle of ensuring a correct protocol split is to verify that both protocols consider all visible states that are reachable by stepping, ensuring a form of progress. Protocols are not required to always terminate and may be used indefinitely, for instance when modeling invariant-based sharing. However, regardless of interleaving or of how many times a shared alias is (consecutively) used, no unexpected state can ever appear in well-formed protocols. Thus, the type information contained in a protocol is valid regardless of all interference that may occur, i.e. it is *stable* [17,32].

Technically, the correctness of protocol splitting is ensured by two key components: 1) a *stepping* relation, that simulates a single use of the shared state through one focus-defocus block; and 2) a *protocol conformance* definition, that ensures full coverage of all reachable states by considering all possible interleaved uses of those steps. Thus, even as the rely and guarantee conditions evolve through the protocol's lifetime, protocol conformance ensures each protocol will never get "stuck" because the protocol must be aware of all possible "alias interleaving" that may occur for that state.

The stepping relation (Fig. 5) uses steps of the form $\langle A, P\rangle \rightarrow \langle A', P'\rangle$ expressing that, assuming shared state A, the protocol P can take a step to shared state A' with residual protocol P'. Due to the use of $\oplus$ and & types in the protocols, there may be *multiple* different steps that may be valid at a given point in that protocol. Therefore, protocol conformance must account for *all* those different transitions that may be picked.

We define protocol conformance as splitting an existing protocol (or capability) in two, although it can also be interpreted as merging two protocols. Regardless of the direction, the actions of the original protocol(s) must be fully contained in the resulting protocol(s). This leads to the three stepping conditions of the definition below.

Definition 1 (Protocol Conformance). Given an initial state A_0 and a protocol γ_0, such protocol can be split in two new protocols α_0 and β_0 if their combined actions conform with those of the original protocol γ_0, noted $\langle A_0 \ , \ \gamma_0 \ \Leftrightarrow \ \alpha_0 \parallel \beta_0 \rangle$. This means that there is a set $\mathcal{S}$ of *configurations* $\langle A \ , \ \gamma \ \Leftrightarrow \ \alpha \parallel \beta \rangle$ closed under the conditions:

1. The initial configuration is in $\mathcal{S}$: $\langle A_0 \ , \ \gamma_0 \ \Leftrightarrow \ \alpha_0 \parallel \beta_0 \rangle \in \mathcal{S}$
2. All configurations take a step, and the result is also in $\mathcal{S}$.
   Therefore, if $\langle A \ , \ \gamma \ \Leftrightarrow \ \alpha \parallel \beta \rangle \in \mathcal{S}$ then:
   (a) exists A', α' such that $\langle A, \alpha \rangle \rightarrow \langle A', \alpha' \rangle$, and for all A', α', $\langle A, \alpha \rangle \rightarrow \langle A', \alpha' \rangle$ implies $\langle A, \gamma \rangle \rightarrow \langle A', \gamma' \rangle$ and $\langle A' \ , \ \gamma' \ \Leftrightarrow \ \alpha' \parallel \beta \rangle \in \mathcal{S}$.
   (b) exists A', β' such that $\langle A, \beta \rangle \rightarrow \langle A', \beta' \rangle$, and for all A', β', $\langle A, \beta \rangle \rightarrow \langle A', \beta' \rangle$ implies $\langle A, \gamma \rangle \rightarrow \langle A', \gamma' \rangle$ and $\langle A' \ , \ \gamma' \ \Leftrightarrow \ \alpha \parallel \beta' \rangle \in \mathcal{S}$.
   (c) exists A', γ' such that $\langle A, \gamma \rangle \rightarrow \langle A', \gamma' \rangle$, and
   for all A', γ', $\langle A, \gamma \rangle \rightarrow \langle A', \gamma' \rangle$ implies either:
   - $\langle A, \alpha \rangle \rightarrow \langle A', \alpha' \rangle$ and $\langle A' \ , \ \gamma' \ \Leftrightarrow \ \alpha' \parallel \beta \rangle \in \mathcal{S}$, or;
   - $\langle A, \beta \rangle \rightarrow \langle A', \beta' \rangle$ and $\langle A' \ , \ \gamma' \ \Leftrightarrow \ \alpha \parallel \beta' \rangle \in \mathcal{S}$.

The definition yields that all configurations must step (i.e. never get stuck) and that a step in one of the protocols (α or β) must also step the original protocol (γ) such that the result itself still conforms. Conformance ensures that all interleavings are coherent. This also means that each protocol "view" of the shared state can work independently in a safe way — even when the other aliases to that shared state are never used. Ownership recovery does not require any special treatment since it just expresses that the focused capability is not returned back to the protocol, enabling it to remain in the local context.

We now apply protocol conformance to our running example, as follows:

$$
\begin{array}{ll}
A : E & \\
\gamma : \textbf{rec}\ X.(\ E \Rightarrow E; X \ \ \& \ \ (\ E \Rightarrow N \oplus C \ ; \ (\ N \Rightarrow \textsf{\textbf{none}} \ \ \oplus \ \ C \Rightarrow \textsf{\textbf{none}}\ )\ )\ ) & \\
\alpha : E \Rightarrow N \oplus C & \text{(Tail protocol)} \\
\beta : \textbf{rec}\ X.(\ E \Rightarrow E; X \ \ \oplus \ \ N \Rightarrow \textsf{\textbf{none}} \ \ \oplus \ \ C \Rightarrow \textsf{\textbf{none}}\ ) & \text{(Head protocol)}
\end{array}
$$

Therefore, applying the definition yields the following set of configurations, $\mathcal{S}$:

$$\langle E \ , \ \textbf{rec}\ X.(\ E \Rightarrow E; X \ \& \ (\ E \Rightarrow N \oplus C; (\ N \Rightarrow \textsf{\textbf{none}} \ \oplus \ C \Rightarrow \textsf{\textbf{none}}\ )\ )\ ) \ \ \Leftrightarrow$$
$$E \Rightarrow C \oplus N \ \parallel \ \textbf{rec}\ X.(\ E \Rightarrow E; X \ \oplus \ N \Rightarrow \textsf{\textbf{none}} \ \oplus \ C \Rightarrow \textsf{\textbf{none}}\ ) \rangle \tag{1}$$

The initial configuration.

by step on γ (subtyping for &) with $E \Rightarrow E; X$ and same with β, using (STEP:ALTERNATIVE-P).

$$\langle N \oplus C \ , \ N \Rightarrow \textsf{\textbf{none}} \ \oplus \ C \Rightarrow \textsf{\textbf{none}} \ \ \Leftrightarrow$$
$$\textsf{\textbf{none}} \ \parallel \ \textbf{rec}\ X.(\ E \Rightarrow E; X \oplus N \Rightarrow \textsf{\textbf{none}} \oplus C \Rightarrow \textsf{\textbf{none}}\ ) \rangle \tag{2}$$

by step on (1) with γ (subtyping for &) with $E \Rightarrow N \oplus C; ...$ and similarly using α.

$$\langle \textsf{\textbf{none}} \ , \ \textsf{\textbf{none}} \ \Leftrightarrow \ \textsf{\textbf{none}} \parallel \textsf{\textbf{none}} \rangle \tag{3}$$

by step on (2) with γ and β using (STEP:ALTERNATIVE-S).

$\mathcal{S}$ is closed (up to subtyping, including unfolding of recursive types).

Regardless of how the use of the state is interleaved at run-time, the shared state cannot reach an unexpected (by the protocols) state. Thus, conformance ensures the stability of the type information contained in a protocol in the face of all possible "alias interleaving". There exists only a finite number of possible (relevant) states, meaning that it suffices for protocol conformance to consider the smallest set of configurations that obeys the conditions above. Since there is also a finite number of possible interleavings resulting from mixing the steps of the two protocols, there are also a finite number of distinct (relevant) steps. Effectively, protocol conformance resembles a form of bisimulation or model checking (where each protocol is modeled using a graph) with a finite number of states, ensuring such process remains tractable.

In the following text we use a simplified notation, of the form $A \Rrightarrow A' \parallel A''$, as an idiom (defined in [21]) that applies protocol conformance uniformly regardless of whether A is a state (for an initial split) or a rely-guarantee protocol (to be re-split and perhaps extended). The missing type is inferred by this idiom.

Example We illustrate these concepts by going back to the pipe's protocols. We introduced the protocols for the head and tail aliases through the share construct:

```
3   share (rw n Empty#[]) as H[n] || T[n];
```

which is checked by the (t:Share) typing rule, using protocol conformance, as follows:

$$\frac{A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \mid \Delta, A_0 \vdash \mathsf{share}\ A_0\ \mathsf{as}\ A_1 \parallel A_2 : [\,] \dashv \Delta, A_1, A_2}\ (\textsc{t:Share})$$

With it we share a capability (A_0) by splitting it in two protocols (A_1 and A_2) whose individual roles in the interactions with that state conform ($\Rrightarrow$). Consequently, the conclusion states that, if the splitting is correct, then in some linear typing environment initially consisting of a type A_0 and Δ, the share construct produces effects that replace A_0 with A_1 and A_2 but leave Δ unmodified (i.e. it is just threaded through).

The next examples show conformance in a simplified way, with only the state and the two resulting protocols of a configuration. Remember that E is the abbreviation for **rw** q `Empty#[]` that, just like the abbreviations C and N, were defined above. Thus, the use of the share construct on line 3 yields the following set of configurations, $\mathcal{S}$:

$$\langle \mathsf{E} \;\Rrightarrow\; \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X\,) \parallel \mathsf{E} \Rightarrow (\, \mathsf{N} \oplus \mathsf{C}\,)\rangle \tag{1}$$

$$\langle \mathsf{N} \oplus \mathsf{C} \;\Rrightarrow\; \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X\,) \parallel \textbf{none}\rangle \tag{2}$$

$$\langle \textbf{none} \;\Rrightarrow\; \textbf{none} \parallel \textbf{none}\rangle \tag{3}$$

The definition is only respected if E is the state to be shared by the protocols. If instead we had shared, for instance, C we would get the next set of configurations:

$$\langle \mathsf{C} \;\Rrightarrow\; \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X\,) \parallel \mathsf{E} \Rightarrow (\, \mathsf{N} \oplus \mathsf{C}\,)\rangle \tag{1}$$

$$\langle \textbf{none} \;\Rrightarrow\; \textbf{none} \parallel \mathsf{E} \Rightarrow (\, \mathsf{N} \oplus \mathsf{C}\,)\rangle \tag{2}$$

The set above does not satisfy our conformance definition. Both the state in configuration (1) and **none** in (2) are not expected by the right protocol. Thus, those configurations are "stuck" and cannot take a step. Although splittings are checked from a high-level and abstracted perspective, their consequences link back to concrete invalid program states that could occur if such invalid splittings were allowed. For instance, in (2), it would imply that the alias that used the right protocol would assume E on focus long after the ownership of that state was recovered by some other alias of that cell. Consequently, such behavior could allow unexpected changes to be observed by that alias, potentially resulting in a program stuck on some unexpected value.

4.3 Using Shared State

Using shared state is centered on two constructs: focus (that exposes the shared state of a protocol) and defocus (that returns the exposed state to the protocol), combined with our version of the *frame rule* (Section 4.4). We now describe how focus is checked:

$$\frac{A_0 \in \overline{A}}{\Gamma \mid A_0 \Rightarrow A_1 \vdash \mathsf{focus}\ \overline{A} : [] \dashv A_0,\ A_1 \triangleright \cdot}\ \text{(t:Focus-Rely)}$$

In general, focus may be applied over a disjunction ($\oplus$) of program states and expected to work on any of those alternatives. By using $\overline{A}$, the programmer can list the types that may become available after focus, nominating what they expect to gain by focus.

focus results in a typing environment where the step of the protocol that was focused on ($A_0 \Rightarrow A_1$) now has its rely type (A_0) available to use. However, it is not enough to just make that capability available, we must also *hide* all other linear resources that may use that same shared state (directly or indirectly) in order to avoid interference due to the inspection of private states. To express this form of hiding, the linear typing environments may include a *defocus-guarantee*. This element, written as $A \triangleright \Delta$, means that we are hiding the typing environment Δ until A is satisfied. Therefore, in our system, the only meaningful type for A is a guarantee type of the form $A';A''$ that is satisfied when A' is offered and enables the protocol to continue to be use as A''. Although the typing rule shown above only includes a single element in the initial typing environment (and, consequently, the defocus-guarantee contains the empty typing environment, $\cdot$), this is not a limitation. In fact, the full potential of (t:Focus-Rely) is only realized when combined with (t:Frame). Together they allow for the non-lexically scoped framing of potentially shared state, where the addition of resources that may conflict with focused state will be automatically nested inside the defocus-guarantee ($\triangleright$). Operationally share, focus, and defocus are no-ops which results in those expressions having type unit ([]).

$$\frac{}{\Gamma \mid \Delta_0,\ A',\ A';A'' \triangleright \Delta_1 \vdash \mathsf{defocus} : [] \dashv \Delta_0,\ A'',\ \Delta_1}\ \text{(t:Defocus-Guarantee)}$$

The complementary operation, defocus, simply checks that the required guarantee type (A') is present. In that situation, the typing environment (Δ_1) that was hidden on the right of $\triangleright$ can now safely be made available once again. At the same time, the step of the protocol is concluded leaving the remainder protocol (A'') in the typing environment. Nesting of defocus-guarantees is possible, but is only allowed to occur on the right of $\triangleright$. Note that defocus-guarantees can never be captured (such as by functions, see Fig. 3 of Section 3) and, therefore, pending defocus operations cannot be forgotten or ignored.

Example We now look at the implementation of the `put` and `close` functions to exemplify the use of focus and defocus. Both functions are closures that capture an enclosing Γ where t is a known location such that `tail` has type **ref** t. T was defined above as: $\forall p.$(**rw** p `Empty#`[] $\Rightarrow$ **rw** p `Node#`R $\oplus$ **rw** p `Closed#`[]) where R is a pair of an integer and a protocol for the head, H (whose definition, given above, is not important here).

```
 9 put = fun( e : int :: rw t exists p.(ref p :: T[p]) ).
                                       Γ = ..., tail : ref t,  t : loc,  e : int | Δ = rw t ∃p.(ref p :: T[p])
10   open <l,last> = new Empty#{} in                 Γ = ..., last : ref l,  l : loc | Δ = ...,  rw l Empty#[]
11     open <o,oldlast> = !tail in              Γ = ...,  oldlast : ref o | Δ = rw t [],  rw l Empty#[],  T[o]
12       focus (rw o Empty#[]);          Δ = ...,  rw o Empty#[],  (rw o Node#R) ⊕ (rw o Closed#[]); none ▹ ·
13       share (rw l Empty#[]) as H[l] || T[l];                                       Δ = ..., T[l], H[l], ...
14       oldlast := Node#{ e, <l,last::H[l]> };                                      Δ = ..., rw o Node#R, ...
15       defocus;                                                                    Δ = rw t [], T[l], none
16       tail := <l, last::T[l]>                                                  Δ = rw t ∃p.(ref p :: T[p])
17     end
18   end,
19 close = fun( _ : [] :: rw t exists p.(ref p :: T[p]) ).
                                         Γ = ..., tail : ref t,  t : loc,  _ : [] | Δ = rw t ∃p.(ref p :: T[p])
20   open <l,last> = !tail in                                  Γ = ..., last : ref l,  l : loc | Δ = rw t [], T[l]
21     delete tail;                                                                                   Δ = T[l]
22     focus (rw l Empty#[]);              Δ = rw l Empty#[],  (rw l Node#R) ⊕ (rw l Closed#[]); none ▹ ·
23     last := Closed#{};                 Δ = rw l Closed#[],  (rw l Node#R) ⊕ (rw l Closed#[]); none ▹ ·
24     defocus                                                                                          Δ = ·
25   end,
```

The `put` function takes an integer stacked with a capability for *t*. The capability is automatically unstacked to *Δ*. Since we are inserting a new element at the end of the buffer, we create a new node that will serve as the new `last` node of that list. On line 11, the `oldlast` node is read from the `tail` cell by opening the abstracted location it contains. Such location refers a protocol type, for which we must use focus (line 12) to gain access to the state that it shares. Afterwards, we modify the contents of that cell by assigning it the new node. This node contains the alias for the new tail as will be used by the head alias. The T component of that split (line 13) is stored in the `tail`. The defocus of line 15 completes the protocol for that cell, meaning that the alias will no longer be usable through there. Carefully note that the share of line 13 takes place *after* focus. If this were reversed, then the type system would conservatively hide the two newly created protocols making it impossible to use them until defocus. By exploiting the fact that such capability is not shared, we can allow it to not be hidden inside ▹ since it cannot interfere with shared state. `close` should be straightforward to understand.

4.4 Framing State

On its own, (T:FOCUS-RELY) is very restrictive since it requires a single rely-guarantee protocol to be the exclusive member of the linear typing environment. This happens because more complex applications of focus are meant to be combined with our version of the frame rule. Together they enable a kind of mutual exclusion that also ensures that the addition of any potentially interfering resources will forcefully be on the right of ▹ (thus making them inaccessible until defocus). The typing rule is as follows:

$$\frac{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \circledast\!\!- \Delta_2 \vdash e : A \dashv \Delta_1 \circledast\!\!- \Delta_2}\ (\text{T:FRAME})$$

Framing serves the purpose of hiding ("frame away") parts of the footprint (Δ_2) that are not relevant to typecheck a given expression (*e*), or can also be seen as enabling extensions to the current footprint. In our system, such operation is slightly more complex than traditional framing since we must also ensure that any such extension will not enable destructive interference. Therefore, types that may refer (directly or indirectly) values that access shared cells that are currently inconsistent due to pending defocus cannot be accessible and must be placed "inside" (on the right of $\triangleright$) the defocus-guarantee. However, statically, we can only make such distinction conservatively by only allowing types that are **non-shared** (and therefore that are known to never conflict with other shared state) to not be placed inside the defocus-guarantee. The formal definition of **non-shared** is in [21], but for this presentation it is sufficient to consider it as pure types, or capabilities (**rw** *p A*) that are not rely-guarantee protocols and that whose contents are also non-shared. This means that all other linear types (even abstracted capabilities and linear functions) must be assumed to be potential sources of conflicting interference. For instance, these types could be abstracting or capturing a rely-guarantee protocol that could then result in a re-entrant inspection of the shared state.

To build the extended typing environment, we define an *environment extension* ($\circledast\!-$) operation that takes into account frame defocus-guarantees up to a certain depth. This means that one can always consider extensions of the current footprint as long as any added shared state is hidden from all focused state. By conservatively hiding it behind a defocus-guarantee, we ensure that such state cannot be touched. This enables locality on focus: if a protocol is available, then it can safely be focused on.

Definition 2 (Environment Extension). Given environments Δ and Δ' we define environment extension, noted $\Delta \circledast\!- \Delta'$, as follows. Let $\Delta = \Delta_n, \Delta_s$ where n-indexed environments only contains ***n*on-shared** elements and s-indexed environments contain the remaining elements (i.e. all those that may, potentially, include *s*haring). Identically, assume $\Delta' = \Delta'_n, \Delta'_s$. Extending Δ with Δ' corresponds to $\Delta \circledast\!- \Delta' = \Delta_n, \Delta'_n, \Delta''_s$ where:

(a) $\Delta''_s = \Delta_{s_0}, A \triangleright (\, \Delta_{s_1} \circledast\!- \Delta'_s \,)$ if $\Delta_s = \Delta_{s_0}, A \triangleright \Delta_{s_1}$
(b) $\Delta''_s = \Delta_s, \Delta'_s$ otherwise.

that either (a) further nests the shared part of Δ' deeper in Δ_{s_1}; or (b) simply composes Δ' if the left typing environment (Δ) does not carry a defocus-guarantee.

Although the definition appears complex, it works just like regular environment composition when Δ' does not contain a defocus-guarantee, i.e. the (b) case. The complexity of the definition arises from the need to nest these structures when they do exist, which results in the inductive definition above. In that situation, we must ensure that any potentially interfering shared state is placed deep inside all previously existing defocus-guarantees, so as to remain inaccessible. This definition is compatible with the basic notion of disjoint separation, but (from a framing perspective) allows us to frame-away defocus-guarantees beyond a certain depth. Such state can be safely hidden if the underlying expression will not reach it (by defocusing).

The definition allows a (limited) form of *multi*-focus. For instance, while a defocus is pending we can create a new cell and share it through two new protocols. Then, by framing the remaining part of the typing environment, we can now focus on one of the new protocols. The old defocus-guarantee is then nested *inside* the new defocus-guarantee that resulted from the last focus. This produces a "list" of pending guarantees in the reverse order on which they were created through focus. Through framing we can hide part of that "list" after a certain depth, while preserving its purpose.

Example We now look back at the focus of line 12. To better illustrate framing, we consider an extra linear type (that is *not* **non-shared**), S, to show how it will become hidden (on the right of ▹) after focus. We also abbreviate the two non-shared capabilities (“**rw** t []” and “**rw** l `Empty#[]`”)[6] as A_0 and A_1, and abbreviate the protocol so that it does not show the type application of location o. With this, we get the following derivation:

$$\dfrac{\dfrac{\dfrac{\mathsf{E} \in \mathsf{E}}{\Gamma \mid \mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C}) \vdash \mathsf{focus}\ \mathsf{E} : [] \dashv \mathsf{E}, (\mathsf{N} \oplus \mathsf{C}); \mathbf{none} \triangleright \cdot}\ (3)}{\Gamma \mid (\mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C})) \circledast\!\!- S, A_0, A_1 \vdash \mathsf{focus}\ \mathsf{E} : [] \dashv (\mathsf{E}, (\mathsf{N} \oplus \mathsf{C}); \mathbf{none} \triangleright \cdot) \circledast\!\!- S, A_0, A_1}\ (2)}{\Gamma \mid \mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C}), S, A_0, A_1 \vdash \mathsf{focus}\ \mathsf{E} : [] \dashv \mathsf{E}, ((\mathsf{N} \oplus \mathsf{C}); \mathbf{none} \triangleright S), A_0, A_1}\ (1)$$

where (1) - (Environment Extension), (2) - (t:Frame), and (3) - (t:Focus-Rely).

Note that frame may add elements to the typing environment that cannot be instantiated into valid heaps. That is, the conclusion of the frame rule states that an hypothesis with the extended environment typechecks the expression with the same type and resulting effects. Not all such extensions obey store typing just like such typing rule enables adding multiple capabilities to one same location that can never be realized in an actual, correct, heap. However, our preservation theorem ensures that starting from a correct (stored typed) heap and typing environment, we cannot reach an incorrect heap state.

4.5 Consumer code

We now show the last function of the pipe example, `tryTake`:

```
26 tryTake = fun( _ [] :: rw h exists p.(ref p :: H[p]) ).                  Δ = rw h ∃p.(ref p :: H[p])
27   open <f,first> = !head in            Δ = rw h [] , (N[f] ⇒ none) ⊕ (C[f] ⇒ none) ⊕ (E[f] ⇒ E[f] ; ...)
           [a] Δ = rw h [], N[f] ⇒ none   [b] Δ = rw h [], C[f] ⇒ none   [c] Δ = rw h [], E[f] ⇒ E[f] ; ...
28     focus C[f], E[f], N[f]; // same abbreviations that were defined above
            [a] Δ = ..., N[f], none; none ▹ ·   [b] Δ = ..., C[f], none; none ▹ ·   [c] Δ = ..., E[f], E[f] ; ... ▹ ·
29     case !first of
30       Empty#_ →                                        [c] Δ = rw h [] , rw f [] , rw f Empty#[]; ... ▹ ·
31         first := Empty#{}; // restore linear type   [c] Δ = rw h [] , rw f Empty#[] , rw f Empty#[]; ... ▹ ·
32         defocus; // the next assignment must occur after defocus and just on this branch   [c] Δ = rw h [] , H[f]
33         head := <f,first::H[f]>;                                              [c] Δ = rw h ∃p.(ref p :: H[p])
34         NoResult#{} : NoResult#([] :: rw h ∃p.(ref p :: H[p])) //assume auto stacked           [c] Δ = ·
35     | Closed#_ →                                                    [b] Δ = rw h [] , rw f [] , none; none ▹ ·
36         delete first;                                                         [b] Δ = rw h [] , none; none ▹ ·
37         delete head;                                                                  [b] Δ = none; none ▹ ·
38         defocus;                                                                                  [b] Δ = ·
39         Depleted#{} : Depleted#[]                                                                 [b] Δ = ·
40     | Node#[element,n] → //opens pair  [a] Δ = rw h [] , rw f [] , n : ∃p.(ref p :: H[p]) , none; none ▹ ·
41         delete first;                              [a] Δ = rw h [] , n : ∃p.(ref p :: H[p]) , none; none ▹ ·
42         head := n;                                            [a] Δ = rw h ∃p.(ref p :: H[p]) , none; none ▹ ·
43         defocus;                                                              [a] Δ = rw h ∃p.(ref p :: H[p])
44         Result#element : Result#(int :: rw h ∃p.(ref p :: H[p])) // assume auto stacked            [a] Δ = ·
45     end
46   end
```

The code should be straightforward up to the use of alternative program states (⊕). This imprecise state means that we have one of several different alternative capabilities

[6] Note that the content of each capability can be made **non-shared** by subtyping rules.

and, consequently, the expression must consider all of those cases separately. On line 28, to use each individual alternative of the protocol, we check the expression separately on each alternative (marked as **[a]**, **[b]**, and **[c]** in the typing environments), cf. (t:Alternative-Left) in Fig. 3. Our `case` gains precision by ignoring branches that are statically known to not be used. On line 29, when the type checker is case analyzing the contents of `first` on alternative **[b]** it obtains type `Closed#[]`. Therefore, for that alternative, type checking only examines the `Closed` tag and the respective case branch. This feature enables the `case` to obey different alternative program states simultaneously, although the effects/guarantee that each branch fulfills are incompatible.

5 Technical Results

Our soundness results (details in [21]) use the next progress and preservation theorems:

Theorem 1 (Progress). *If e_0 is a closed expression (and where Γ and Δ are also closed) such that* $\Gamma \mid \Delta_0 \vdash e_0 : A \dashv \Delta_1$ *then either:*

- *e_0 is a value, or;*
- *if exists H_0 such that $\Gamma \mid \Delta_0 \vdash H_0$ then $\langle\, H_0 \parallel e_0 \,\rangle \mapsto \langle\, H_1 \parallel e_1 \,\rangle$.*

The progress statement ensures that all well-typed expressions are either values or, if there is a heap that obeys the typing assumptions, the expression can step to some other program state — i.e. a well-typed program never gets stuck, although it may diverge.

Theorem 2 (Preservation). *If e_0 is a closed expression such that:*

$$\Gamma_0 \mid \Delta_0 \vdash e_0 : A \dashv \Delta \qquad \Gamma_0 \mid \Delta_0 \circledast\!\!- \Delta_2 \vdash H_0 \qquad \langle\, H_0 \parallel e_0 \,\rangle \mapsto \langle\, H_1 \parallel e_1 \,\rangle$$

then, for some Δ_1 and Γ_1 we have: $\Gamma_0, \Gamma_1 \mid \Delta_1 \circledast\!\!- \Delta_2 \vdash H_1 \qquad \Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A \dashv \Delta$

The theorem above requires the initial expression e_0 to be closed so that it is ready for evaluation. The preservation statement ensures that the resulting effects (Δ) and type (A) of the expression remains the same throughout the execution. Therefore, the initial typing is preserved by the dynamics of the language, regardless of possible environment extensions ($\circledast\!\!- \Delta_2$). This formulation respects the intuition that the heap used to evaluate an expression may include other parts (Δ_2) that are not relevant to check that expression.

We define *store typing* (see [21]), noted $\Gamma \mid \Delta \vdash H$, in a linear way so that each heap location must be matched by some capability in Δ or potentially many rely-guarantee protocols. Thus, no instrumentation is necessary to show these theorems.

Destructive interference occurs when an alias assumes a type that is incompatible with the real value stored in the shared state, potentially causing the program to become stuck. However, we proved that any well-typed program in our language cannot become stuck. Thus, although our protocols enable a diverse set of uses of shared state, these theorems show that when rely-guarantee protocols are respected those usages are safe.

6 Additional Examples

We now exemplify some sharing idioms captured by our rely-guarantee protocols.

6.1 Sharing a Linear ADT

Our protocols are capable of modeling monotonic [12,24] uses of shared state. To illustrate this, we use the linear stack ADT from [22] where the stack object has two possible typestates: Empty and Non-Empty. The object, with an initial typestate E(mpty), is accessible through closures returned by the following "constructor" function:

$$
\begin{aligned}
!(\ \forall \mathtt{T}.\ [\,] \multimap \exists \mathtt{E}.\exists \mathtt{NE}.\ ![\quad \mathtt{push} &: \mathtt{T} :: \mathtt{E} \oplus \mathtt{NE} \ \multimap\ [\,] :: \mathtt{NE},\\
\mathtt{pop} &: [\,] :: \mathtt{NE} \multimap \mathtt{T} :: \mathtt{E} \oplus \mathtt{NE},\\
\mathtt{isEmpty} &: [\,] :: \mathtt{E} \oplus \mathtt{NE} \multimap \mathtt{Empty}\#([\,] :: \mathtt{E}) + \mathtt{NonEmpty}\#([\,] :: \mathtt{NE}),\\
\mathtt{del} &: [\,] :: \mathtt{E} \multimap [\,]\ ]\ \ ::\ \mathtt{E}\ )
\end{aligned}
$$

Although the capability to that stack is linear, we can use protocols to share it. This enables multiple aliases to that same object to coexist and use it simultaneously from unknown contexts. The following protocol converges the stack to a non-empty typestate, starting from an imprecise alternative that also includes the empty typestate.

$$(\mathtt{NE} \oplus \mathtt{E}) \Rightarrow \mathtt{NE}\ ;\ \textbf{rec}\ X.(\mathtt{NE} \Rightarrow \mathtt{NE}\ ;\ X)$$

Monotonicity means that the type becomes successively more precise, although each alias does not know when those changes occurred. Note that, due to focus, the object can undergo intermediate states that are not compatible with the required NE guarantee. However, on defocus, clients must provide NE such as by pushing some element to the stack. The protocol itself can be repeatedly shared in equal protocols. Since each copy will produce the same effects as the original protocol, their existence is not observable.

6.2 Capturing Local Knowledge

Although our types cannot express the same amount of detail on local knowledge as prior work [4, 18], they are expressive enough to capture the underlying principle that enables us to keep increased precision on the shared state between steps of a protocol.

For this example, we use a simple two-states counter. In it, N encodes a number that may be zero and P some positive number, with the following relation between states:

$$\mathsf{N} \triangleq \mathsf{Z}\#[\,] + \mathtt{NZ}\#\mathsf{int} \qquad \mathsf{P} \triangleq \mathtt{NZ}\#\mathsf{int} \qquad \text{(note that: } \mathsf{P} <: \mathsf{N}\text{, vital to show conformance)}$$

We now share this cell in two asymmetric roles: IncOnly, that limits the actions of the alias to only increment the counter (in a protocol that can be shared repeatedly); and Any, an alias that relies on the restriction imposed by the previous protocol to be able to capture a stronger rely property in a step of its own protocol. Assuming an initial capability of rw p N, this cell can be shared using the following two protocols:

$$
\begin{aligned}
\mathtt{IncOnly} &\triangleq \textbf{rec}\ X.(\ \mathsf{rw}\ p\ \mathsf{N} \Rightarrow \mathsf{rw}\ p\ \mathsf{P}\ ;\ X\ )\\
\mathtt{Any} &\triangleq \textbf{rec}\ Y.(\ \mathsf{rw}\ p\ \mathsf{N} \Rightarrow \mathsf{rw}\ p\ \mathsf{P}\ ;\ \mathsf{rw}\ p\ \mathsf{P} \Rightarrow \mathsf{rw}\ p\ \mathsf{N}\ ;\ Y\ )
\end{aligned}
$$

Thus, by constraining the actions of IncOnly we can rely on the assumption that Any remains positive on its second step, even when the state is manipulated in some other unknown program context. Therefore, on the second step of Any, the case analysis can be sure that the value of the shared state must have remained with the NZ tag between focuses. Note that the actions of that alias allow for it to change the state back to Z.

6.3 Iteratively Sharing State

Our technique is able to match an arbitrary number of aliases by splitting an existing protocol. Such split can also extend the original uses of the shared state by appending additional steps, if those uses do not destructively interfere with the old assumptions.

This example shows such a feature by encoding a form of delegation through shared state that models a kind of "server-like process". Although single-threaded, such a system could be implemented using co-routines or collaborative multi-tasking. The overall computation is split between three individual workers (for instance by each using a private list containing cells with pending, shared, jobs) each with a specific task. A Receiver uses a Free job cell and stores some Raw element in it. A Compressor processes a Raw element into a Done state. Finally, the Storer removes the cells in order to store them elsewhere. In real implementations, each worker would be used by separate handlers/threads, triggered in unpredictable orders, to handle such jobs.

We also show how we can share multiple locations together, bundled using $*$, by each job being kept in a *c*ontainer cell while the *f*lag (used to communicate the information on the kind of content stored in the container) is in a separate cell. The raw value is typed with A and the processed value has type B. The types and protocols are:

$$\mathsf{F} \triangleq \mathbf{rw}\ f\ \mathtt{Free\#[]}\ *\ \mathbf{rw}\ c\ [] \qquad \mathsf{R} \triangleq \mathbf{rw}\ f\ \mathtt{Raw\#[]}\ *\ \mathbf{rw}\ c\ A \qquad \mathsf{D} \triangleq \mathbf{rw}\ f\ \mathtt{Done\#[]}\ *\ \mathbf{rw}\ c\ B$$

$$\begin{aligned} \mathsf{Receiver} &\triangleq \mathsf{F} \Rightarrow \mathsf{R} \\ \mathsf{Compressor} &\triangleq \mathbf{rec}\ X.(\ \mathsf{F} \Rightarrow \mathsf{F}; X \ \oplus\ \mathsf{R} \Rightarrow \mathsf{D}\ ) \\ \mathsf{Storer} &\triangleq \mathbf{rec}\ X.(\ \mathsf{F} \Rightarrow \mathsf{F}; X \ \oplus\ \mathbf{rec}\ Y.(\ \mathsf{R} \Rightarrow \mathsf{R}; Y \ \oplus\ \mathsf{D} \Rightarrow \textsf{\textbf{none}}\ )\ ) \end{aligned}$$

The protocol for the Receiver is straightforward since it just processes a free cell by assigning it a raw value. Similarly, Compressor and Storer follow analogous ideas by using a kind of "waiting" steps until the cell is placed with the desired type for the actions that they are to take (note how Storer keeps a more precise context when the state is not F, even though it is not allowed to publicly modify the state). To obtain these protocols through binary splits, we need an *intermediate* protocol that will be split to create the Compressor and Storer protocols. The initial split (of F) is as follows:

$$\mathsf{F} \ \Rrightarrow\ \mathsf{Receiver}\ \|\ \mathbf{rec}\ X.(\ \mathsf{F} \Rightarrow \mathsf{F}; X \ \oplus\ \mathsf{R} \Rightarrow \textsf{\textbf{none}}\ )$$

The protocol on the right is then further split, and its ownership recovery step further extended with additional steps, to match the two new desired protocols:

$$\mathbf{rec}\ X.(\ \mathsf{F} \Rightarrow \mathsf{F}; X \ \oplus\ \mathbf{rec}\ Y.(\ \mathsf{R} \Rightarrow \mathsf{R}; Y \ \&\ \mathsf{R} \Rightarrow \mathsf{D}; \mathsf{D} \Rightarrow \textsf{\textbf{none}}\ )\ ) \ \Rrightarrow\ \mathsf{Compressor}\ \|\ \mathsf{Storer}$$

The Receiver alias never needs to see how the other two aliases use the shared state. Although the second split is independent from the initial one, protocol conformance ensures that it cannot cause interference by breaking what Receiver initially relied on.

7 Related Work

We now discuss other works that offer flexible sharing mechanisms. Although there are other interesting works [1,2,4,5,7,31] in the area, they limit sharing to an invariant.

In *Chalice* [19], programmer-supplied permissions and predicates are used to show that a program is free of data races and deadlocks. A limited form of rely-guarantee is used to reason about changes to the shared state that may occur between atomic sections. All changes from other threads must be expressed in auxiliary variables and be constrained to a two-state invariant that relates the *current* with the *previous* state, and where all rely and guarantee conditions are the same for all threads.

Several recent approaches that use advanced program logics [9, 10, 23, 30, 32] employ rely-guarantee reasoning to verify inter-thread interference. Although our approach is type-based rather than logic-based, there are several underlying similarities. *Concurrent abstract predicates* [9] extend the concept of *abstract predicates* [23] to express how state is manipulated, supporting internally aliased state through a *fiction of disjointness* (also present in [16, 18]) that is based on rely-guarantee principles and has similarities to our own abstractions. Their use of rely-guarantee also allows intermediate states within a critical section, which are immediately weakened (made stable) to account for possible interference when that critical section is left. Although our use of rely-guarantee is tied to state (be it references or abstracted state), not threads, our protocols capture an identical notion of stability through a simpler constraint that ensures all visible states are considered during protocol conformance. Another modeling distinction is that our interference specification lists the resulting states (from interference), not the actions that can (or cannot [10]) occur from external/unknown sources.

Monotonic [12, 24] based sharing enables unrestricted aliasing that cannot interfere since the changes converge to narrower, more precise, states. Our protocols are able to express monotonicity. However, since the rely and guarantee types of a step in the protocol must describe a finite number of states, we lack the type expressiveness of [24]. We believe this concern is orthogonal to our core sharing concepts, and is left as future work. We are also capable of expressing more than just monotonicity. For instance, due to ownership recovery, a cell can oscillate between shared and non-shared states during its lifetime, and with each sharing phase completely unrelated to previous uses.

Gordon *et al.* [15] propose a type system where references carry three additional type components: a predicate (for local knowledge), a guarantee relation, and a rely relation. They handle an unknown number of aliases by constraining the writes to a cell to fit within the alias' declared guarantee, similarly to how rely-guarantee is used in program logics to handle thread-based interference. Although they support a limited form of protocol (and their technique can generally be considered as a two-state protocol), their system effectively limits the actions allowed by each new alias to be strictly decreasing since their guarantee must fit within the original alias' guarantee. Since we support ownership recovery of shared state, a cell can be shared and return to non-shared without such restriction. Unlike ours, their work does not allow intermediate inconsistent states since all updates are publicly visible. In addition, their work requires proof obligations for, among other things, guarantee satisfaction while we use a more straightforward definition of protocol conformance that is not dependent on theorem-proving. However, their use of dependent refinement types adds expressiveness (e.g. their predicates capture an infinite state space, while our state space is finite) but increases the challenges in automation, as typechecking requires manual assistance in Coq.

Krishnaswami *et al.* [18] define a generic sharing rule based on the use of frame-preserving operations over a commutative monoid (later shown to be able to encode rely-guarantee [8]). The core principle is centered on splitting the internal resources of an ADT such that all aliases obey an invariant that is shared, while also keeping some knowledge about the locally-owned shared state. By applying a frame condition over its specification, their shared resources ensure that any interference between clients is benign since it preserves the fiction of disjointness. Thus, local assumptions can interact with the shared state without being affected by the actions done through other aliases of that shared state. The richness of their specification language means that although it might not always be an obvious, simple or direct encoding, protocols are likely encodable through the use of auxiliary variables. However, our use of a protocol paradigm presents a significant conceptual distinction since we do not need sharing to be anchored to an ADT. Therefore, we can share individual references directly without requiring an intermediary module to indirectly offer access to the shared state, but we also allow such uses to exist. Similarly, although both models allow ownership recovery, our protocols are typing artifacts which means that we do not need an ADT layer to enable this recovery and the state of that protocol can be switched to participate in completely unrelated protocols, later on. Their abstractions are also shared symmetrically, while our protocols can restrict the available operations of each alias asymmetrically. Additionally, after the initial split, our shared state may continue to be split in new ways. Finally, we use focus to statically forbids re-entrant uses of shared state, while they use dynamic checks that diverge the execution when such operation is wrongly attempted.

8 Conclusions

We introduced a new flexible and lightweight interference control mechanism, *rely-guarantee protocols*. By constraining the actions of an alias and expressing the effects of the remaining aliases, our protocols ensure that only benign interference can occur when using shared state. We showed how these protocols capture many challenging and complex aliasing idioms, while still fitting within a relatively simple protocol abstraction. Our model departs from prior work by, instead of splitting shared resources encoded as monoids, offering an alternative paradigm of "temporal" splits that model the coordinated interactions between aliases. A prototype implementation, which uses a few additional annotations to ensure typechecking is decidable, is currently underway[7].

Acknowledgments. This work was partially supported by Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under grant SFRH / BD / 33765 / 2009 and the Information and Communication Technology Institute at CMU, CITI PEst-OE / EEI / UI0527 / 2011, the U.S. National Science Foundation under grant #CCF-1116907, "Foundations of Permission-Based Object-Oriented Languages," and the U.S. Air Force Research Laboratory. We thank the Plaid (at CMU) and the PLASTIC (at UNL) research groups, and the anonymous reviewers for their helpful comments.

References

1. A. Ahmed, M. Fluet, and G. Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 2007.

[7] Available at: https://code.google.com/p/deaf-parrot/

2. N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA*, 2008.
3. N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *ECOOP*, 2011.
4. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, 2007.
5. L. Caires and J. a. C. Seco. The type discipline of behavioral separation. In *POPL*, 2013.
6. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. Logic in Computer Science*, 2007.
7. R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, 2004.
8. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL*, 2013.
9. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.
10. M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.
11. M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, 2002.
12. M. Fähndrich and K. R. M. Leino. Heap monotonic typestate. In *IWACO*, 2003.
13. S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, 2010.
14. J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 1987.
15. C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *PLDI*, 2013.
16. J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, 2012.
17. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 1983.
18. N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *ICFP*, 2012.
19. K. R. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, 2009.
20. Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP*, 2003.
21. F. Militão, J. Aldrich, and L. Caires. Rely-guarantee protocols (technical report). CMU-CS-14-107, 2014.
22. F. Militão, J. Aldrich, and L. Caires. Substructural typestates. In *PLPV*, 2014.
23. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, 2005.
24. A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *TLDI*, 2011.
25. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. Logic in Computer Science*, 2002.
26. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proc. LISP and Functional Programming*, 1992.
27. F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP*, 2000.
28. R. E. Strom. Mechanisms for compile-time enforcement of security. In *POPL*, 1983.
29. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 1986.
30. K. Svendsen, L. Birkedal, and M. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, 2013.
31. J. A. Tov and R. Pucella. Practical affine types. In *POPL*, 2011.
32. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.
33. G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manipulating programs. *Electron. Notes Theor. Comput. Sci.*, 2005.

Networking and Information Technology
Research and Development

NITRD and SCORE Workshop Series

Designing-In Security: Current Practices and Research Needs

July 1-2, 2013
Software Engineering Institute
4401 Wilson Boulevard, Suite 1000, Arlington VA 22203

WORKSHOP REPORT

Contributors:

Celia Merzbacher, Ronald Perez, William Scherlis, Tomas Vagoun, Claire Vishik, Angelos Keromytis, Lisa Coote, et. al.

Table of Contents

Executive Summary

In addressing the need for improved Designed-In Security (DIS) research and practice, the Federal Networking and Information Technology Research and Technology[1] (NITRD) Program and the Special Cyber Operations Research and Engineering Interagency Working Group[2] (SCORE IWG) have begun conducting a series of small, invitational workshops with the aim of placing leading researchers in direct contact with leading practitioners to ensure that future research targets those underlying problems that truly limit the practice. NITRD and SCORE have adopted a multidisciplinary approach whereby a broad range of experts are included in the workshops to address the various problems and solutions that may have previously gone overlooked as a consequence of an overly narrow focus. The workshops offer an opportunity for practitioners to become more familiar with research concepts to address their current needs and, similarly, for academics to gain familiarity with operational challenges as well as better identify educational needs and approaches in building a workforce capable of designing and producing higher assurance systems. The innovative ideas identified in the first workshop will be developed and evaluated in subsequent workshops of the series.

Introduction

Trustworthy Cyberspace: Strategic Plan for the Federal Cybersecurity Research and Development Program[3] prioritized Designed-In Security (DIS) as a research theme to foster research that:

> *Builds the capability to design, develop, and evolve high assurance, software-intensive systems predictably and reliably while effectively managing risk, cost, schedule, quality, and complexity. Promotes tools and environments that enable the simultaneous development of cyber-secure systems and the associated assurance evidence necessary to prove the system's resistance to vulnerabilities, flaws, and attacks. Secure, best practices are built inside the system. Consequently, it becomes possible to evolve software-intensive systems more rapidly in response to changing requirements and environments.*

[1] http://www.nitrd.gov

[2] SCORE IWG is the U.S. Government's forum for coordinating cybersecurity research activities related to national security systems.

[3] http://www.nitrd.gov/SUBCOMMITTEE/csia/Fed_Cybersecurity_RD_Strategic_Plan_2011.pdf, National Science and Technology Council, December 2011.

Identifying research objectives requires an understanding of current problem areas, successes, and lessons learned by developers of high assurance software and hardware systems. Of critical importance is understanding what factors limit our current state of practice as well as what we can infer about the future of Designed-In Security.

The “Designed-In Security: Current Practices and Research Needs” workshop was held July 1-2, 2013 at the Software Engineering Institute in Arlington, Virginia. The workshop focused on the IT hardware and software sectors, and posed the following questions to the participants:

- What procedures are in use in your industry now for designing in security?
- What processes do you use to identify and validate the best practices in use or that are contemplated for use in your organization?
- What approaches for Designed-In Security in, beyond those currently in use, would you advocate are ready for industry adoption?
- For each such practice, what is the evidence (in terms of effectiveness, resource cost, scalability, usability, and other criteria) to support its use?
- What hard research problems are in most urgent need of solutions?

The workshop opened with leading practitioners discussing the current state of the practice in DIS, followed by a discussion by leading researchers on the state of the research in DIS. After the opening discussion, the participants were organized into three groups: Business Case, Software, and Hardware. The following sections provide summaries of the discussions and the breakout groups.

State of the Practice

The workshop began with two presentations on the state of the DIS practice by Steve Lipner (Microsoft) and Mary Ellen Zurko (Cisco). Both presenters noted that it has not been until recently that commercial vendors have worked towards developing system architectures that enhance security attributes. Efforts remain hampered by a number of factors and a lack of concrete, transferable metrics that have prevented decision makers from shifting their organization towards a more DIS-centric approach. One such factor is the challenge of integrating security-related activities into modern development processes, particularly agile processes. At larger scales, architecture design is a more explicit activity, with a stronger reliance on the technical capability and professional reputation of security architects. In contrast, at the small scale, agile processes—as currently implemented—may tend to de-emphasize the importance of up-front consideration of architecture, thereby thwarting secure design.

Another factor that plays a large role in how strongly DIS is emphasized is the perspective of the customer. Customers with a higher regard for security are more likely to place the same level of emphasis on the security requirements as they do on the functional requirements; thereby conveying Designed-In Security as a project deliverable to the vendor. However, many customers who have longstanding relationships with their suppliers work under the assumption that the appropriate security features and characteristics will be inherently included by the vendor in the final product. As a consequence, discussions concerning particular security attributes, measurement of these attributes, and the consequences of incidents are relatively limited. Furthermore, standards and compliance regimes tend to focus on process compliance and checklists that outline particular known attacks. As a result, it is difficult to justify a significant increment of cost or delay when customers cannot readily assess the benefits in a direct way, even when providers and clients have a strong trust relationship.

As researchers and practitioners begin designing future security technologies, heightened visibility for security and advanced measurement capabilities (to improve the capacity to assess security attributes of systems during development) are critical in persuading those who are capable of influencing widespread DIS adoption. In assuring security attributes early on, it is also necessary for both the designer and user to understand the operating environment as well as such factors as the potential hazards, potential vulnerabilities, nature of the threats, and the technical characteristics of the software and hardware components of the system. With these considerations in mind, designers will be able to identify security-related technologies and interventions that can support a business case for adoption.

In the past decade, Microsoft advanced an aggressive agenda of reworking its development practices and achieved widespread adoption of these practices by internal development groups. Recognizing that an early focus on security can be greatly beneficial, Microsoft took explicit steps to advance security-related interventions as early as possible in the development lifecycle, as is evident from the Security Development Lifecycle[4] (SDL) process framework. Microsoft was not only responding to the challenge of how to design "good code to begin with," but also the economic consequences of handling defects late in the lifecycle.

[4] *The Security Development Lifecycle*, M. Howard & S. Lipner, Microsoft Press, May 2006.

Among Microsoft's interventions, the SDL has perhaps had the greatest impact on industry practices outside of Microsoft, as is evident from published Building Security In Maturity Model[5] (BSIMM) results.

At Microsoft, choices of programming language are seen as important drivers of outcomes. In particular, deficiencies in C and C++ are largely responsible for a growing industry focused specifically on code-level defects and vulnerabilities. Language improvements, such as strong typing, which advanced into mainstream languages twenty years ago, can make a difference. In order to be more effective in both software design and code synthesis as well as analysis of code and artifacts, improvements must be made to the methods and tools used by the development teams in conducting these tasks. This is necessary for both products and services, in the sense that software is a service.

With respect to the processes and the timing of interventions, advances such as SDL are becoming generally accepted and adopted as a reference model. According to practitioners, models, such as SDL, provide a comprehensive identification of security-related activities during development, as well as experience-based guidance regarding which activities to undertake at which points in a development process. SDL addresses a broad range of threats related to *S*poofing, *T*ampering, *R*epudiation, *I*nformation disclosure, *D*enial of service, and *E*levation of privilege, otherwise known at Microsoft as STRIDE. The BSIMM assessment, developed and administered by Gary McGraw and collaborators, evaluates organizations by comparing their actual practices with those best practices identified by models such as SDL. However, despite the emerging consensus regarding identified best practices, there is nonetheless relatively little science or data to support specific claims regarding security outcomes. This is yet another factor contributing to the "measurement gap" that needs to be addressed with further research.

With respect to tooling, practitioners recognize that the growing variety of process-support tools is, in fact, beneficial and also necessary for nearly all development and sustainment teams. These include Integrated Development Environments (IDEs) for individual developers such as Eclipse, Visual Studio, Rational tools, etc., as well as team tools that support functions such as configuration management of artifacts, automated testing and analysis, issue tracking, and other critical activities. While impact may not be as profound as in process interventions, tooling and analysis also remain encumbered by the "measurement gap." The "measurement

[5] The BSIMM is a study of real-world software security initiatives, http://bsimm.com/

gap” for tooling is exacerbated by the diversity of security related technical quality attributes as well as by the imprecise results of tools used to support analysis and testing.

In education, the challenges concerning DIS range from those encountered by novice end-users confronted with phishing and other social-engineering attacks, to seasoned engineers facing challenges with tooling, analysis, architecture, processes, requirements, and other considerations.

Integrating Practice into Large Organizations

In recognizing the benefits of best practices such as Software Development Life Cycle (SDLC) or SSDL and Microsoft’s SDL, the question then becomes how best to integrate these practices into one’s organization. Current methodologies are primarily concerned with prescriptive models, unique to each company, that stipulate exactly what needs to be done to ensure the best practices are being fully engaged. As an effective alternative, the creators of BSIMM have created a descriptive model describing what is actually happening in one’s organization rather than what should be happening. BSIMM is a descriptive model that can be used to measure any number of prescriptive SSDLs. In essence, BSIMM is a measuring stick capable of applying 122 different measurements to 62 real security initiatives. The underlying idea of BSIMM is to build a maturity model from actual data gathered from 9 well known large-scale software security initiatives. Once the maturity model has been validated by data gathered from 62 different companies, statistical analysis will be performed to determine how effective the model is, which activities correlate with each other, and whether or not high maturity firms appear at similar levels. In time, more activities will be added with changes in security. This tool allows not only a comparison of individual companies to one another but also a comparison of groups/types of companies. Through applying this tool, it was revealed that most companies proceed in the same or similar ways.

State of the Research

The workshop’s State of the Research panel was led by John Launchbury (Galois) and included as panelists J.R. Rao (IBM), Michael Reiter (University of North Carolina), Robert Seacord (SEI), and Elaine Weyuker (formerly Rutgers). Panelists currently serve as leading researchers in areas related to Designed-In Security (DIS), including both technical and empirical dimensions, across the spectrum from basic science to assessment of industrial adoption.

The discussion started with an exploration of barriers to industry adoption of advanced technologies and practices related to DIS. Historically, this has been a significant issue, with persistent difficulties related to Return on Investment (ROI), scaling, analytic accuracy, and performance. Difficulties in calculating ROI relate to difficulties in measurement and also the credibility of claims regarding the value of newly emerging techniques. ROI difficulties also derive from the perceived incremental cost and delay of attending more aggressively to security issues, when the returns on the cost increments are uncertain and difficult to measure. Scaling has been a historical difficulty with both technical and process approaches. Analytic accuracy relates, for example, to the rates of false positives in static analysis tools. Performance relates both to the impact of using advanced tooling on the efficiencies of individual and team practices and also to the overhead associated directly with using the tool, such as from dynamic analyses. Against the background of these historical difficulties is a sense of optimism on all four of these fronts. Much of the discussion at the workshop centered on new research advances, transition successes, and identified transition success patterns that provide evidence in support of this sense of optimism.

Several panelists pointed out the benefits of taking explicit steps to bridge the gap between research and practice, transferring research results to practice when they have overcome barriers of realism, for example, and also transferring from practice to researchers a better understanding of the real problems to be solved and the important acceptance criteria for these problems. Recurring challenges include both size and complexity scaling, barriers in usability by ordinary developers, steepness of learning curve for new technologies and practices, early demonstrations of benefit to technology adopters, and the challenges of making a compelling business case even when other challenges are overcome.

There is a perception of an unavoidable tradeoff between the level of capability and the extent of assured security – that features must be sacrificed in order to ensure a higher level of quality and security. In the discussion, a counterpoint emerged, suggesting that as quality and security practices improve, over the long haul, functionality can actually be increased due to the greater expressiveness of safer high-level abstractions, including understandable security policy concepts. This means moving away from a “penetrate and patch” model and towards security by design, and taking this attitude in multiple dimensions of practice, ranging from audit and logging to project planning and incident response procedures. It also includes looking closely at the multiple components that comprise large systems and identifying mission critical assets, access paths to those assets, and constructing “fine grained perimeters,” such as those that might come through secure processors and root-of-trust architectures.

One example of a transition success and an associated success pattern is the work related to secure coding practices for mainstream languages, including C, C++, and Java. By adopting certain identified concrete coding structures in common situations where naively constructed structures might be dangerous, practicing developers can realize security benefits without much effort. This work has also influenced the ISO C standards process, which has adopted certain small language changes that help developers to more naturally write secure code.

Breakout Sessions

In transitioning to more focused discussions, participants separated into their respective groups of Business Case, Software, and Hardware. Each group was challenged with questions concerning how current practices have come into being; the effects of current practices on stakeholders; limitations of current practices and the possible improvements to those practices; the necessary incentives in achieving those improvements; and the effects of changes on stakeholders.

Breakout Group: Business Case

The Business Case group focused on elements of the business case for DIS as well as research and policies to increase adoption in various sectors. A widely recognized benefit of DIS is that it is less expensive to build in security early on in the design process rather than later in the process or even after the product is deployed in the field. However, in general, difficulties in computing Return on Investment (ROI) has prevented DIS from being implemented on a large scale. Rather than trying to estimate ROI, one participant recommended that companies think of security investment as “insurance” against an increasingly likely and damaging breach. Of course, insurance only pays off if there is a perceived threat. In the Y2K example, companies treated those modifications effected as “insurance,” however, after January 2000 those efforts came to a halt.

Participants also identified specific factors that influence a business’s decision to pursue DIS. By far the most influential factor is customer demand, which has historically been relatively weak. For the most part, building in security is perceived by the customer as a “cost of doing business" rather than as an "attribute" or "feature" that contributes to reliability and quality. This is exacerbated by the measurement difficulties discussed above. As security almost never leads to a standalone product security enhancements have not gained the recognition that other, more lucrative features enjoy. However, casting security as an attribute may make its development cost more palatable. Recognition and growing concern of IP theft/loss and the discovery of various side-channel vulnerabilities are motivating the development of security strategies. While changes in government regulations and requirements are also driving investment in security, it was pointed out that government business alone is unlikely to drive industry since it encompasses only a small fraction of most companies' business. Transforming corporate culture into one that supports "security" requires efforts both from top down and bottom up.

A potential trend that could improve security decision making is the movement towards vertical integration in some sectors, whereby more layers of the hardware-software stack are controlled by one company (e.g. Apple's control of hardware and software design). Greater vertical integration allows security to be more targeted and more likely to be cost effective.

The lack of metrics and standards for security also hampers the business case for investment. Standards can be driven by new technology, but typically take time to be developed and adopted. While some claim that security is too subjective to allow for deterministic metrics; the notion of measuring "processes" related to security is generally perceived as both feasible and necessary.

In identifying what research is needed to help form a better business case for DIS in both hardware and software, research concerning techniques that can reduce the cost/time of Designing-In Security was identified as particularly necessary with the caveat that such research should be informed by best practices for incorporating security in different sectors. Advanced research could also be directed towards developing a BSIMM-type model that is more quantitative and scalable, rather than using strictly qualitative processes such as interviews.

In addition to technical research, economic/business research is needed to develop "predictive models" that anticipate the economic impact of security outcomes. Economic models for security should be developed to enable research in creative business models, similarly to what happened with the research concerning the Economics of the Internet in the early 90's. These models will enable developers and businesses to predict the economic outcomes of introducing security features, including requirements for and impact on civil infrastructure. Such research includes developing a large scale simulation of global systems and "systems of systems.” However, in order to support such multidisciplinary research, a common language of agreed-upon security-related definitions and well-defined taxonomies/ontologies are needed.

Other potential research topics that will support the business case for DIS include research into risk/resilience analysis; security and emerging “usage environments” such as social networks; Bring Your Own Device (BYOD); trust relationships among suppliers and customers; and overarching principles that embody economic and security objectives, similar to those developed in the area of “Green Chemistry.”

With regards to technology transfer issues, current university research is viewed as insufficiently generalizable and therefore impractical in real systems, although novel ideas can be used internally in the organization’s R&D programs. In some cases, a startup can be successful in moving research from the university through the early development phase and, in these instances, venture capital typically plays an important role. However, security technologies are usually supplemental to core features in products, with the consequence that venture capital and other support for incubation and start-up phases are typically weaker than in other technology areas. Despite these weaknesses, often the greatest value of university research as perceived by the private sector is access to faculty and student expertise.

In addition to recommendations of future research areas, participants discussed strategies and policies that could further strengthen the DIS business case. Leveraging existing government-funded programs such as the Small Business Innovation Research (SBIR) and In-Q-Tel were identified as possible mechanisms for facilitating technology transfer. The NSF Science of Innovation and Science Policy program could also support the economic research that is needed to support innovation and the business case for security. The Enduring Security Framework brings together high level industry and government decision makers and could also be leveraged to create high level agreement to support DIS. Building a “community” around secure hardware and software design could be strengthened by an annual workshop similar to the “Economics of Information Security” workshop series. The introduction of such a forum could stimulate increased industry-government-academia collaboration to address both pre-competitive and non-competitive challenges.

Breakout Group: Software

The two primary sets of questions that became the focus of the software working group included: identifying and evaluating current best practices, with a heightened emphasis on (1) what's working well, (2) where research is needed, and (3) what are patterns of success in adopting development practices and tools that can materially improve security outcomes, and secondly, identifying opportunities and challenges, with a purpose of identifying (1) areas of significance to practice where technical progress could be accelerated, (2) hard problems that offer game-change possibilities but that will require some sustained attention, and (3) mechanisms to accelerate technology transition.

Practice: Requirements. One of the first issues that came up in the discussion was related to security requirements, and specifically how potential security-related requirements can be specified and assessed, beyond expressions such as "keep the bad guys out" or "be able to handle what comes our way." The challenge is how to identify and express particular requirements and then be able to assess their appropriateness to the purpose and operating environment of the system being developed or evolved. The question, in essence, is how we can achieve early validation for particular requirement specifications and models. This is a notable challenge since the threat, operational, and infrastructural environments are constantly evolving and, furthermore, the challenge is exacerbated because many (indeed, most) of the critical technical attributes are not readily testable. Economic drivers are very significant during requirements setting—which for many/most commercial systems is a continuous process over the lifetime of a system. Due to challenges in creating valid economic models, it is often very difficult to navigate the engineering trade-offs other than on the basis of informal experience and expert judgment.

Practice: SDL and Similar Models. The Security Development Lifecycle (cited above) has become a significant positive influence on the culture of development organizations. It incorporates process milestones that include support of modeling and direct evaluation of development artifacts. While the principal focus is on internal measures of process compliance, there are also emerging informal external measures. This is a very significant area where more research is needed. Also, as noted above, the BSIMM evaluation framework defines various key process areas for normative best practice. This enables organizations to assess the extent of their adoption of identified best practices in comparison with identified norms. Although the model is broadly considered to be highly valuable, the connections with results are still informally constructed due to the ongoing challenges of inadequate risk methodology and, more generally, the "measurement gap" for software security attributes and underlying quality attributes.

As discussed in other groups and in plenary, the group emphasized the difficulty of making an effective quantitative business case for security-related interventions in the process of development and evaluation. Due to the “measurement gap,” interventions are accepted by senior management in many cases on the basis of technical experience and expertise rather than on the basis of predictive quantitative business models. On the issue of training and education, critical to success is an understanding of the scope and diversity of security-related skills, ranging from system administration activities such as device and platform configuration to architectural design and evaluation. Another issue is how advanced methods are transitioned into practice across a larger enterprise. Most often, there are central teams that capture experience, develop models of costs, benefits, and risks, and can assist internal development organizations in advancing their practice in a way that harmonizes with their problem domain and team culture.

Practice: Modeling and Analysis. The group spent some time discussing particular technical interventions related to modeling, analysis, language, development/evolution data, and architecture. Most significantly, there is a recognition in industry, perhaps most visibly at Microsoft, of the significant value of advanced technical interventions in all these areas—and undertaking those interventions as early as possible in the process.

Technical models and associated mathematical methods for various kinds of analysis are becoming more broadly adopted for a growing set of quality attributes contributing to security outcomes. It is now recognized that a wide variety of models are needed, as is a corresponding variety of analysis methods with varying degrees of formality and rigor. Models are used in requirements formulation, architecture, design, and coding; they are used to predict outcomes with respect to functional features, quality attributes including security, performance, and many other characteristics. Success in any engineering discipline relies on a range of models that are expressive, that cover critical functional and quality attributes, and that can be effectively validated.

The rigor and scalability of analyses associated with particular models can increase naturally over time, with the advancement of the underlying models and, where possible, foundational mathematics. As research proceeds, models become more expressive and useful, while the associated analyses may advance from informal representations, such as diagrams and documents, to formally-based analyses that work in small-scale cases to better analyses that are composable, scalable, and computationally feasible for large systems. This is evident when considering a full range of analytic techniques, model checking, static analysis, type checking, and verification. The ability to accommodate a range of informal and formal techniques for modeling and analysis is important for success in practice.

As models and tools are developed, there will be a natural advance in the development of appropriate metrics to assess impact despite the fact that there is a considerable lag of metrics behind technical developments and interventions. This suggests that senior managers have come to rely more directly on expert technical judgment in making choices regarding interventions and supporting those choices with some kind of business case. For this reason, the most aggressive adoption is in technology-intensive firms with technology-savvy senior management who understand the issues behind the lag. This highlights the more general issue of how costs, benefits, and risks are assessed for security-related interventions; there is a general perception that some interventions, in the long run, will likely improve productivity in development and evaluation, and sometimes dramatically. However, there are risks and costs associated with any new technology adoption. This highlights the importance of crafting new practices and tools in ways that they can be readily assimilated into practice with minimal incremental cost and risk.

Practice: Programming Language. The choice of programming language is significant. This is, in fact, counter to early folklore that quality outcomes are primarily due to management and process interventions rather than technological interventions. The pervasive modern recognition is that management and process interventions are enabled and enhanced using technology, and that the two approaches are increasingly intertwined. Each language choice, whether it is C, C++, C#, JavaScript, or PHP, has costs, risks, and benefits, and these must be weighed in the context of overall development goals. Additionally, evaluations of these languages must take into consideration not only the technical characteristics of the particular language, but also the associated socio-technical ecosystem, including tools, frameworks and libraries, and human programmer resources.

Associated with languages are identified models and patterns, which include certain "secure coding" practices. Many of these practices have been codified, and these best practices have been explicitly adopted as enterprise standards in technology firms such as Oracle, Cisco, and Siemens. In addition, the practices have influenced the natural evolution of the formal language-definition standards, such as the C-language standards from ISO.

Practice: Architecture. Decisions regarding overall architecture for software-reliant systems are critical. Processes for making and validating architecture choices are both an essential feature of success and a dark (and proprietary) art—the "secret sauce"—in the commercial world. These choices have tremendous leverage on outcomes related to both quality attributes and sustainability/modernization. A good architecture, for example, can localize and minimize the critical portions of the system that require the largest amount of attention in development and evaluation—enabling a lesser standard to be more safely applied to the larger remaining portions of the system.

Complicating these choices, besides the intrinsic technical difficulties of modeling and analysis, are a range of extrinsic factors including legacy and precedent, domain culture, and organizational and supply-chain structure. Also complicating these choices are architectural design choices, such as those related to resiliency and "moving target" concepts that are directly motivated by the presence of potential adversaries. These include not just adaptation and shape-shifting, but also sensing to detect attacks and configuration damage.

Practice: Data. Contributing to the advancement of all these areas is the dramatic increase in the extent of granular data associated with the software development and evolution processes. This is largely a consequence of the advancement of tool technology, as noted above, and it affords great opportunity to better address the "measurement gap" as well as to afford a possibility of mitigating information loss in development and assuring better control over the configuration integrity of systems and associated artifacts (what Microsoft calls "Managed Code"). Of course, the gap is a moving target, in the sense that technical advancement to close that gap on one end occurs in parallel with improvements in technology and practices that further open the gap on the other end.

Research: Evidence Production. After several decades of slow advancement, there is now more rapid progress and optimism in the research community in a number of areas including languages, tools, models, analysis, architecture, and usability. These advances are enabled by progress in disciplines ranging from foundational mathematics to tool design and human social psychology.

One of the principal research challenges has been the massive information loss that occurs in the development and evolution of complex software-reliant systems. This results in enormous costs in the evaluation of systems, which can involve extensive reverse engineering to rediscover design abstractions and rationale, as well as in sustainment and modernization activities, for similar reasons. The prospects of evidence-based approaches, where evidence in support of security-related claims is amassed during development and sustained in configuration integrity, has been greatly enabled by the advance of modern tooling, modeling, and analysis, all of which support the creation and capture of massive amounts of development data, as well as supporting the configuration management of this data through the lifecycle as systems and associated models co-evolve.

Research: Usability. It is now recognized that usability by humans—as end-users, as essential parts of an overall system, and as developers and evaluators—must be a major consideration in the advancement of practices. For end users, the design of security-related abstractions must result in metaphors that can be embraced by end users. Certain end user populations can be trained to assimilate the new metaphors, but for others this is not readily possible. For developers, the design of languages, APIs, tools, and models are all influenced by usability considerations. Usability is not the same as simplicity—complexity can be accepted if it can be encapsulated, such as in type systems, or if there is sufficient support by tools and models to help people in managing it.

Research: Hard Problems. The working group recognized some challenges in particular that are in need of sustained attention. Among these challenges include those concerning architecture design, modeling, and analysis. In organizations, there is still a dominant "guru model" where security-critical areas of a system are isolated through canny architectural choice-making, and then handled by experts. A second area of hard problems relates to requirements, and particularly requirements associated with security-related attributes, such as STRIDE, as mentioned earlier. Thirdly, there are the challenges of today's ubiquitously rich supply chains—how to ensure security in systems that are developed by contractors working within arm's-length of the main mission stakeholder.

Research: Technology Transition. The technology transition challenge has several dimensions, one being the advancement of empirical software engineering and science-of-security research to better support the evaluation and validation of hypotheses, including those associated with the claimed benefits of new methods, practices, and tools. A second is measurement, more generally, identified above as the problem of the "measurement gap." Progress in this area would not only facilitate technology transition, but also the management of complex supply chains. A third challenge for R&D managers is tracing the impact of early research investments on outcomes in practices. Several participants noted that the transition pathways are complex and often diffuse, thwarting traceability despite a recognition of the essential role of advances in basic science in achieving major advances the practice. In response to this, the R&D community has undertaken several studies to document both the pathways and significance of the R&D role. These are more comprehensively described in a series of several studies from the National Research Council.

Breakout Group: Hardware

The hardware breakout group was predominantly represented by industry and academic experts in software and/or systems engineering who had a particular focus in the semiconductor industry. In this regard, the question of how to apply abstraction approaches from the software community to hardware community was one of the first questions posed by participants. As a recommendation for future workshops, participants recommended for the continuation of identifying current limitations of Designed-In Security in the hardware space. Participants argued that DIS has not been more successful because (1) the Return on Investment (ROI) is difficult to calculate and therefore unclear; (2) security is not thought of as a core element of the product; (3) there is a lack of clarity regarding government initiatives; and (4) there is a lack of standard vocabulary/taxonomy. Participants also pushed for addressing these current limitations by exploring high visibility government/industry sponsored challenges and best paper competitions. Topics should be narrowed to a specific problem and addressed by a handful of stakeholder representatives to enable a gap-analysis of specific domains. With regards to format, workshop proceedings should consist of more topic-focused tracks with subject matter expert/session leaders who are all given the opportunity to prepare for the workshop in advance.

One recommendation for easing the introduction of security mechanisms in the hardware was to develop them as dual-use components that have some other useful functionality. Typically, such mechanisms might improve reliability, such as through isolation, or provide debugging or performance monitoring capabilities, such as with the Last Branch Record register or micro-architecture counters.

Regarding recommendations for the state of practice, the group began by endorsing greater diversity from the hardware community, but without sacrificing the software representation since software experience and perspective is always necessary. There is serious potential for improving overall system security through greater collaboration between hardware and software industries. Specifically, there is need for more intense co-design among the hardware and software industries, particularly at the early stages of the software development process. For example, a technique already apparent in the “verticalization trend” is compressing the hardware development cycle so that it better aligns with that of the software development life cycle. Additionally, techniques and approaches from the software security community are potentially less complex, more bounded and easily applied in the hardware domain. Information flow/taint analysis, composability, abstraction, etc. were all provided as examples.

As a benefit to the hardware industry, research is relatively mature in terms of overall design and transition of research to practice. While security is far from reaching the maturity level of other hardware aspects it would be beneficial to leverage the maturity of hardware to formally define hardware-related security properties and specifications that could drive security verification and other tooling. Hardware aspects to be leveraged include architecture specification, high and low-level design, quality/reliability, verification, widespread use of sophisticated tools, and formal analysis. Participants also observed that current research and best practices are generally isolated to security specific features, capabilities, solutions, and communities such as smart cards and hardware security module ecosystems. There is need for a hardware focused security engineering community, and best practices that are emphasized at all phases of development. Referred to as ***Design for Security,*** participants called for developing hardware equivalents of SDL, BSIMM, etc., and leveraging opportunities to draw from some of the hardware discipline’s unique insights such as behavioral analysis, anomaly detection, runtime attestation-like capabilities, authentication, and provenance. The workshop group called for more widespread application of hardware capabilities, to existing challenges and initiatives such as building secure systems from less/unsecure components, moving target defense, tailored trustworthy spaces, reference monitors, etc. The need for further examination and exploration of hardware in vertical markets was also mentioned, with particular emphasis on those markets in which there is a greater need for security such as medical/healthcare, cyber physical systems, and critical infrastructure.

Conclusions

The overall goal for the Designed-In Security Workshop (DIS) is a clear recipe for how we can adapt and enhance the practices in current use for development, sustainment, operation, and evolution of systems, with a goal of supporting an integrated approach to "Designed-In" assurance. This "Designed-In" approach has the potential not only to support more rapid evaluations, but also much higher levels of assurance for complex software-reliant systems. Importantly, this approach lays the foundation for rapid re-certification as systems evolve and are further interconnected with other systems. However, successfully achieving such re-certification efforts requires a firm grasp of the necessary metrics for evaluating future systems. In reflecting upon which insights would pose the greatest value towards a new approach, sector leaders identified the workshop recommendations they found to be most critical. From the software sector, one of the great benefits of modern tools, and an area where there is opportunity for further advancement, is the retention and exploitation of the large amounts of data associated with modern software production. This data can support the advancement of metrics as well as the production of evidence in support of assurance cases. From the hardware perspective, in addition to more intense co-design with the software sector, leaders highlighted the need for developing security mechanisms as dual-use components that provide additional functional capability. And finally, drawing on a more holistic perspective, business case leaders emphasized the need for a forum that includes industry and government agencies, where a set of diverse, multidisciplinary researchers can share results and ideas, analogous to the annual Workshop on Economics of Information Security.

Contact Information

For more information about this workshop, contact:

National Coordination Office for Networking and Information Technology R&D
4201 Wilson Blvd., Suite II-405, Arlington, VA 22203
Phone: 703-292-4873
Email: nco@nitrd.gov

Appendix A: 2013 Designed-In Security Workshop Agenda

JULY 1, 2013

8:00 am Registration

8:30 – 8:45 Introduction and goals for the workshop

8:45 – 10:15 State of the Practice

Talks by industry representatives laying out the processes in use in their firms to develop a particular product/system and how and where security considerations influenced the design process.

- Steve Lipner, Microsoft
- Mary Ellen Zurko, Cisco

10:15 – 10:30 Break

10:30 – 12:00 State of the Research

Panel by leading researchers on recent results bearing on methods for designing in security, including empirical evidence obtained or needed to support industrial adoption.

- John Launchbury, Galois, Panel Chair
- J.R. Rao, IBM
- Michael Reiter, UNC
- Robert Seacord, SEI
- Elaine Weyuker, Rutgers

12:00 – 1:00 Lunch break

1:00 – 2:30 Breakout Groups: Best Practices

- Breakout Group: Software (Lead: Bill Scherlis, SEI)
- Breakout Group: Hardware (Lead: Ron Perez, AMD)
- Breakout Group: Business Case (Lead: Celia Merzbacher, SRC)

Each group tasked to identify:

- Current best practices for designing in security
- How the practices have come into being
- Effects of the current practices on stakeholders
- What limits current practices, where they might be improved
- What incentives might be required to achieve the improvement
- Effects of changes on stakeholders

2:30 – 2:45 Break

2:45 – 3:30 Plenary Talk

- Gary McGraw, Cigital

3:30 – 4:30 Breakout Groups: Research

Breakout groups continue, shifting to research focus, to identify:

- What research results are available that might advance best practices?
- What evidence is available that these methods would improve practice?
- What research problems are suggested by the forgoing discussions?
- How do you make assessments of the potential of the practice to scale with acceptable risk and cost?

4:30 – 5:15 Breakout Groups: Progress Report

Brief in-progress reports from each of the breakout groups.

JULY 2, 2013

8:30 – 8:45 Introduction to second day

8:45 – 10:15 Breakout Groups: Summary

Breakout groups develop summary characterizations of industry best practices and research agendas.

10:15 – 10:30 Break

10:30 – 12:00 Breakout Groups: Brief-outs

Brief outs by the groups and discussion.

12:00 pm Adjourn

Appendix B: Workshop Participants

Organizing Committee

Name	Position	Organization
Martin, Brad	Committee Chair	NSA
Landwehr, Carl	Research Consultant	
Newhouse, Bill	National Initiative for Cybersecurity Education (NICE) Program Lead, Cybersecurity R&D Coordination	NIST
Scherlis, Bill	Professor & Director, Institute for Software Research (SCS/ISR), School of Computer Science	CMU/SEI
Vagoun, Tomas	Cybersecurity R&D Coordinator	NCO/NITRD
Vishik, Claire	Security & Privacy Technology & Policy Manager	Intel

Invitees

Name	Position	Organization
Adam, Nabil	Distinguished Professor of Computer Information Systems and Director of Rutgers CIMIC Research Center	Rutgers University
Aitken, Rob	R&D Fellow	ARM
Dill, Stephen	LM Fellow, Center for Cyber Security Innovation	Lockheed Martin
Elder, Matthew	Sr. Manager, Development, Symantec Research Labs	Symantec
Fogerson, Tim	Security Engineering Manager	Intel
Green, Cordell	Director and Chief Scientist	Kestrel Institute
Jaeger, Trent	Professor, Computer Science and Engineering	Pennsylvania State University
Keromytis, Angelos	Associate Professor, Computer Science Department	Columbia University
Kirby, James	SW Engineering Researcher	Navy Research Laboratory
Launchbury, John	Chief Scientist	Galois
Lipner, Steve	Partner Director of Program Management, Trustworthy Computing	Microsoft
McGraw, Gary	CTO	Cigital
Merzbacher, Celia	Vice President, Innovative Partnerships	SRC
Ostrand, Tom	Visiting Scholar, Center for Discrete Mathematics and Theoretical Computer Science & AT&T Labs	Rutgers University

Ozkaya, Ipek	Senior Member of Technical Staff, Architecture Practices	SEI
Perez, Ron	Senior Fellow, Senior Director, Security Architecture Organization	AMD
Rajan, Anand	Manager, Security Research Lab	Intel
Rao, Josyula	Director of Security Research	IBM Research
Reiter, Mike	Professor, Department of Computer Science	University of North Carolina
Seacord, Robert	Secure Coding Team Lead	SEI
Tinnel, Laura	Senior Research Engineer	SRI International
Totah, John	Technical Director in the Office of the CTO	Oracle
van Doorn, Leendert	Corporate Fellow, Corporate VP	AMD
Wagner, Grant	Technical Director, Trusted Systems Research Group	NSA
Weyuker, Elaine	Visiting Scholar, Center for Discrete Mathematics and Theoretical Computer Science & AT&T Labs	Rutgers University
Zurko, Mary Ellen	Security Architect and Strategist	Cisco